

High-Level Control of Microprocessor Components

An Approach Based on an Adaptable
Compiler System for Modula-2

ERIC ASTOR



LUND UNIVERSITY

Department of Computer Sciences

1984



Parts of this work have been conducted within a project supported by the National Swedish Board for Technical Development (STU 81-3425).

High-Level Control of Microprocessor Components
=====

-
An Approach based on an Adaptable Compiler System for Modula-2
=====

Eric Astor

CODEN: LUNFD6/(NFCS-1001)/1-125/(1984)

Department of Computer Sciences
Lund University
Box 725
S-220 07 LUND
Sweden

Institutionen för Datateknik
Datalogi och Numerisk Analys
Lunds Universitet
Box 725
220 07 LUND

TABLE OF CONTENTS

CHAPTER -----	PAGE -----
INTRODUCTION	1
1 PROBLEM DISCUSSION	3
1.1 LOW-LEVEL FACILITIES IN MICROPROCESSOR SYSTEMS	3
1.2 PROGRAMMING LANGUAGES FOR LOW-LEVEL CONTROL	7
1.3 OBJECTIVES OF THE THESIS	9
1.3.1 Utilization and control demands	9
1.3.2 Additional demands	10
2 THE ADAPTABLE COMPILER SYSTEM	11
2.1 PRESENTATION	11
2.1.1 The basic idea	11
2.1.2 The BOBS parser generator	12
2.1.3 The description part	14
2.1.4 The front end part	15
2.1.5 The code generating part	17
2.2 RELATED WORK	18
2.2.1 Table-driven code generation techniques	18
2.2.2 Similarities and differences	20
2.2.3 Notational conventions	22
3 IMPLEMENTATION-DEPENDENT ASPECTS OF MODULA-2	23
3.1 STANDARD OBJECTS	23
3.1.1 Standard data types	23
3.1.2 Standard operators	27
3.1.3 Standard procedures	31
3.2 SYSTEM OBJECTS	32
3.2.1 System data types	32
3.2.2 System procedures	33
3.3 TYPE COMPATIBILITY	34
3.4 THE SYSTEM MODULE	35

TABLE OF CONTENTS (CONT)

CHAPTER	PAGE
4 COMPONENT MODULE SPECIFICATIONS	36
4.1 TYPE SPECIFICATIONS	37
4.1.1 Standard type specifications	37
4.1.2 System type specifications	38
4.2 OPERATOR SPECIFICATIONS	43
4.3 PROCEDURE SPECIFICATIONS	46
4.3.1 Standard procedure specifications	46
4.3.2 System procedure specifications	48
4.4 SPACE ALLOCATION	52
5 THE DESCRIPTION MODULES	54
5.1 THE IMPLEMENTOR MODULE	55
5.1.1 Object declarations	56
5.1.2 Attribute declarations	56
5.1.3 Terminal symbols	57
5.1.4 Grammar and code rule specifications	58
5.1.5 Run-time procedures	58
5.2 THE COMPONENT MODULE	60
5.3 THE CONFIGURATION MODULE	64
6 THE INTERMEDIATE REPRESENTATION	66
6.1 ADDRESSING	66
6.1.1 Address modes	66
6.1.2 Allocation areas	67
6.1.3 Prefixes	67
6.1.4 Access paths	68
6.2 STRUCTURED ELEMENTS	71
6.3 PROCEDURE CALLS	72

TABLE OF CONTENTS (CONT)

CHAPTER	PAGE
7 RUN-TIME ORGANIZATION	74
7.1 DESIGN CONSIDERATION	74
7.1.1 Run-time routines	74
7.1.2 Linked references	75
7.1.3 Storage areas	75
7.2 RUN-TIME ORGANIZATION IN THE iAPX86	76
7.2.1 Local storage disposition	76
7.2.2 System storage disposition	78
8 SPECIFICATION OF A NUMERIC COPROCESSOR	80
8.1 TYPE SPECIFICATIONS	80
8.2 OPERATOR SPECIFICATIONS	80
8.2.1 Synchronization	83
8.3 PROCEDURE SPECIFICATIONS	84
8.3.1 Special NDP instructions	85
8.3.2 Error handling	86
9 IMPLEMENTATION OF PROCESS CONTROL	87
9.1 PROCESS CONTROL OPERATIONS IN THE iAPX86	87
9.1.1 Implementation of NEWPROCESS	88
9.1.2 Implementation of TRANSFER	89
9.2 PROCESS CONTROL OPERATIONS IN THE iAPX286	90
9.2.1 Implementation of NEWTASK	93
9.2.2 Implementation of TASKTRANSFER	95
10 IMPLEMENTATION OF INTERRUPT CONTROL	96
10.1 INTERRUPT CATEGORIES	96
10.2 IMPLEMENTATION OF IOTRANSFER	97
10.3 PRIORITY INTERRUPTS	102

TABLE OF CONTENTS (CONT)

CHAPTER	PAGE
11 MULTI-PROCESSING CONTROL	104
11.1 IMPLEMENTATION OF CRITICAL REGIONS	105
11.1.1 The iAPX86 family	105
11.1.2 The Z8000 family	106
11.1.3 The M68000 family	107
11.2 IMPLEMENTATION OF SEMAPHORES	108
12 PROTECTION AND VIRTUAL MANAGEMENT CONTROL	110
12.1 SIMPLE PROTECTION MECHANISMS	110
12.2 PROTECTION AND MEMORY MANAGEMENT	111
12.2.1 The Z8010 Memory Management Unit	111
12.2.2 Protected Virtual Address Mode in the iAPX286	113
13 UTILIZATION OF AN OPERATING SYSTEM COMPONENT	119
14 CONCLUSIONS	121
14.1 SUMMARY	121
14.2 DISCUSSION	122
14.3 LIMITATIONS AND FUTURE WORK	123
14.4 PRACTICAL APPLICATIONS	123
14.5 STATISTICS	124

ACKNOWLEDGEMENTS

REFERENCES

APPENDIX The Syntax of the ACS

INTRODUCTION

The purpose of this work is to demonstrate that it is possible to control and to utilize low-level facilities implemented by the components of a microprocessor system directly from a high-level programming language without the requirement of assembler coded routines.

The approach taken is to organize an adaptable compiler system for the systems programming language Modula-2 [Wirt82]. Given specifications concerning the implementation of data types and operations supported by the components in the actual configuration, the compiler system will regenerate the code generating part and make relevant information available to the front end part of the Modula-2 compiler. With the compiler adapted in this way the programmer will be fully supported to utilize and control low-level facilities such as operand precision, process control, internal and external interrupts, and coprocessor functions.

The compiler system is based on the BOBS parser generator [Erik82]. However, the choice of the parser generator is not of primary importance. The YACC parser writing system [John75] was also available as an alternative. Both systems accept LALR(1) grammars as input and generate optimized parser tables as output. The translation technique used by the Modula-2 compiler originates from the work of Graham-Glanville [Glan78, Grah80] on retargetable code generators based on pattern-matching table-driven methods. The BOBS system offers a simple but straightforward way of including synthesized attributes in the translation scheme that will be used in a similar way as presented in the work of Ganaphati [Gana80, Gana82].

The thesis concentrates on the components of the iAPX86 microprocessor family. The reasons for this are, first, that the ideas and design decisions presented will need a consistent framework to be related to and exemplified from. Second, the iAPX86 family will cover most of the basic facilities found in the other microprocessor families. In the later parts of the thesis implementation details and special components from other families will also be treated to some extent.



The thesis is organized as follows:

CHAPTER 1 discusses the problem addressed. Different kinds of low-level facilities of interest to control and to utilize are presented. CHAPTER 2 gives an overview of the adaptable compiler system. The BOBS parser generator is presented together with the other parts of the compiler system. The influences from related work are discussed. CHAPTER 3 introduces the programming language Modula-2 from the implementors perspective.

CHAPTER 4 presents the component module and defines the specifications for standard and system data types, operators and procedures. The definitions are exemplified by specifications for the iAPX86 processor.

CHAPTER 5 presents the implementor module and the configuration module. The specifications of terminals, attributes, code rules, run-time procedures and support objects are defined.

CHAPTER 6 gives an insight of the intermediate representation. The notation for operand addressing, control statements, procedure calls and other operations is described.

CHAPTER 7 discusses the run-time organization in the iAPX86 micro-processor system.

CHAPTER 8 demonstrates the specification of a numeric data processor.

CHAPTER 9 discusses the implementation of process control exemplified by the specification of the TRANSFER operation in two ways; simple process control in iAPX86, and hardware supported task control in iAPX286.

CHAPTER 10 explains the different types of interrupts present in most microprocessors and discusses their implementation. The problem of priority interrupts is discussed.

CHAPTER 11 shows how the multi-processing support offered by some processors can be integrated in high-level constructs by use of system procedures.

CHAPTER 12 covers the control of protection and virtual memory management in some processors.

CHAPTER 13 describes the utilization of a firmware component from the Modula-2 level.

CHAPTER 14 concludes the thesis with a summary of the results and gives some figures regarding the compiler system and the code generated.

A compiler system, written in Pascal, realizing the ideas in this work has been partially implemented on a VAX 780/11 computer with the results transferred to and tested in a microcomputer system.

CHAPTER 1. PROBLEM DISCUSSION

The very large scale integration techniques has spurred a component development in mainly two directions. One direction points toward highly specialized components designed for a specific purpose. The other direction points toward single chip computers including a general purpose processor together with memory, and peripheral interfaces in one component. The specialized components are designed to work together with other components in different types of arrangements making up more or less unique component configurations. The single chip computers offer fixed arrangements of commonly used configurations. In both cases there are demands for a programming language that combines the virtues of a modern high-level language with the abilities of assembly languages of low-level control and utilization of the facilities present in new components and changing configurations.

It is the intention of this thesis to demonstrate that the high-level language Modula-2 will stand up to these demands when supported by a compiler system with the ability to adapt to various low-level requirements.

In order to give an orientation of the problems involved the next section will give a survey of low-level facilities and, when applicable, the constructs to deal with them in Modula-2.

1.1 LOW-LEVEL FACILITIES IN MICROPROCESSOR SYSTEMS

Multiple data type support

Modula-2 only defines one basic precision of the numeric data types, while most processors implement arithmetics for more than one precision of numeric operands. In order to take advantage of the arithmetic abilities offered by a processor the compiler writing system should support the implementation of an optional number of precisions for each basic data type. If numeric coprocessor components are included in a configuration the motivation for this kind of support is further strengthened.

Utilization of special instructions

Many processors include special instructions that are of interest to utilize directly from the programming level. Examples of such

instructions are block move instructions, status control, special test instructions, and many others. From the high-level programmers point of view instructions of this kind should be referenced as procedures, but on the implementation level the desired instruction(s) rather than a procedure call should be generated.

Coprocessor utilization

The use of coprocessors in a configuration combines the possibilities, and problems, of extended data type support and utilization of special instructions. Coprocessors also introduce the problem of supporting the communication and synchronization interface between the coprocessors and the main processor.

Process transfer control

The Modula-2 TRANSFER operation corresponds to a very basic coroutine transfer and is intended as a primitive to be used to implement more sophisticated and safe process handling operations programmed at the Modula-2 level. The compiler writing system should allow different implementation strategies and separate implementations for different kinds of process concepts. Especially it should allow the implementation to take advantage of hardware supported process types and process transfer mechanisms.

Interrupt transfer control

The Modula-2 interrupt processing is based on the IOTRANSFER operation that will bind the specified interrupt code to the associated interrupt process. In addition it specifies a second process variable that will receive as value the process executing when the interrupt occurs.

The compiler writing system should allow the resulting Modula-2 compiler either to generate straight code to implement the interrupt mechanism, or to generate an interface to a run-time routine. It should also support the implementation of separate interrupt mechanisms. For instance external and internal interrupts may be processed in different ways, or more than one process type is to be supported by the implementation. The implementation of the interrupt mechanism in a real-time system requires more than maybe any other implementation decision to be optimally tuned to the application.

Interrupt priority control

The priority control mechanisms differ among the micro processor families. The most customary among processors with built-in priority control is to support a strict hierarchical priority scheme. But there are also processors that allow more complicated priority structures. Some processors do not support priority level interrupts by themselves but assume that this is taken care of by an external interrupt controller. The external interrupt controller is often a very capable component that can be programmed to support a broad range of interrupt policies.

The generated Modula-2 compiler should be able to conform to the interrupt priority control supported by the processor, including the control of different interrupt policies supported by the hardware.

Multi-processing support

Mechanisms for multi processing support are needed in computer systems where more than one processor may be accessing a shared resource simultaneously. Most microprocessors provide very basic mechanisms for multi-process applications that must be further supported by the hardware.

The Modula-2 implementation should allow the programmer to integrate the existing multi-processing hardware mechanisms in software constructs to implement different kinds of semaphores, conditional regions, monitors and other abstractions of importance in concurrent programming.

Protection and virtual memory management control

Many processors are able to execute in a protected mode where certain privileged instructions cannot be executed and certain reserved memory areas cannot be accessed. Protection of this kind is important in operating systems or in other multi-tasking systems where kernel functions and data must be safe against illegal or unintentional use. As Modula-2 is a systems programming language intended for programming such systems the protection mechanisms should be under control from the Modula-2 level.

Memory protection is a more versatile protection mechanism than the protection mode control. It is often implemented by specialized memory management components that compare all memory references against programmed tables or descriptors specifying protected memory areas. The protection may be aimed at reading, writing, execution, or simply prohibiting any reference.

A virtual memory manager expands the memory space seen by the programmer by using a combination of directly accessible memory and peripheral storage devices. As a result, the programmer can write large programs without worrying about the physical memory limitations.

The Modula-2 programmer should be able to take full advantage of such mechanisms, i.e. from the controlling point of view.

Firmware components

Firmware components implement special operations encoded in non-volatile memory, optionally supported by integrated hardware supported facilities. An example of such operations is a collection of primitives for an operating system kernel. The problem of utilizing firmware components often comes down to a calling convention. The compiler system should support direct access to the operations and facilities provided.

Port programmed components

Controllers and other components programmed via fixed ports require that the ports are possible to declare by absolute address references in the Modula-2 program. In case the processor supports two or more separate address spaces a notation to specify the address space where the port is located will be required as well.

Some of the port programmed components require command sequences where the timing between the commands may be crucial. In such cases delay loops with controlled lengths must be inserted between the commands.

1.2 PROGRAMMING LANGUAGES FOR LOW-LEVEL CONTROL

Assemblers

The traditional programming languages for low-level control are the assembler languages. The advantage of using assembler languages for this purpose is obvious as the facilities implemented by the hardware components are designed to be controlled from the assembler level. The disadvantage is that assembler languages do not fulfill criteria such as security, readability and portability. These basic criteria become increasingly more important the bigger, the more complex, and the more expensive a system is.

Structured assemblers incorporating elementary high-level control and data structures are more readable than pure assemblers. The translation required to convert a program written in a structured assembler language is in most cases straightforward and may even be performed by macro expansion. The high-level statements typically remain in the listing as comments, and most systems allow modifications to the generated assembler statements without having to recompile the source program. Thus structured assemblers offer more readability and, to some small extent, more security than pure assemblers, while the scope of low-level control remains.

Medium-level languages

The term medium-level language can be applied to those languages which combine higher-level concepts with the possibility for low-level control at the cost of security. Examples of languages belonging to this category are C [Kern78] and Forth [Moor74]. The main difference between these languages and proper high-level languages is that in the high-level languages the low-level constructs are designed as a part of the language, while in the medium-level languages the low-level control is obtained as a result of a general freedom due to the lack of restricting rules.

High-level languages

High-level languages for low-level control can roughly be divided into three categories.

- a) Dedicated machine-dependent high-level languages designed to work in a particular hardware environment. Languages in this category are

often derived from PL360 [Wirt68] designed to allow full use of the IBM 360 hardware. Other examples are 'in house made' systems programming languages used for internal software development for a particular computer system and not generally available.

b) General high-level languages extended with machine dependent low-level constructs not originally present in the language. The most used base languages in this category are Pascal [Jens75], and subsets of PL/I [IBM 68]. Examples of such extensions are absolute addressing facilities, directives to allocate data and code at fix locations, interrupt procedures, inline assembler code, access to certain processor registers and more. The combination of a language of this category with a structured assembler is especially favoured by manufacturers of development systems for microprocessors.

c) General high-level languages incorporating low-level constructs in the original design with means to isolate machine-dependent parts. Examples of this category are more recent high-level languages such as Ada [Ichb80] and Modula-2. Both languages support absolute allocation, controlled suspension of type compatibility, and include constructs for process and interrupt handling. Ada is the conceptually more powerful and comprehensive language also at the low level, while the basic design philosophy of Modula-2, i.e. simplicity, is apparent also at the low level. The process primitives in Modula-2 are intended rather for building user-defined constructs than for immediate use, while the more complicated Ada task mechanisms are intended for use in the form they are defined in the language.

Summary

Of the languages discussed in the this section only the assemblers and machine-dependent high-level languages will qualify for use in micro component configurations where high demands for control and utilization are present. If we also insist on using newly developed components with added facilities then it will be hard to keep up with a repeated expansion and maintainance of a traditional compiler for a machine-dependent high-level language and only the assembler alternative remains. This is a situation that holds back the software development for new components and interferes with an advantageous utilization of the possibilities offered by advanced hardware facilities.

1.3 OBJECTIVE OF THE THESIS

The objective of the thesis is to demonstrate that the problem of advanced low-level control from a modern high-level language can be solved by the support of an adaptable compiler system.

The high-level language chosen is Modula-2. The reason for this is that Modula-2 has convenient features for isolating machine dependencies together with basic low-level control facilities. Ada would also be a feasible candidate in this respect but the task of writing even an experimental compiler system for this much more complex language seems less attractive.

1.3.1 Utilization and control demands

With reference to the previous survey of low-level facilities in microprocessor components it is the intention to make Modula-2 capable of handling the following utilization and control problems.

1) Utilization of multiple precision data types.

In addition to the basic data types; INTEGER, CARDINAL, CHAR, REAL, BOOLEAN, enumeration and SET, other precisions of these types should be possible to define in order to utilize existing hardware support. Operations for such additional data types should be specified in terms of code sequences for the supporting components, i.e. the main processor or numeric coprocessors.

2) Utilization of special functions.

It should be possible to take advantage of special functions supported by coprocessor components or firmware components by a high-level procedure call translated to straight code realizing the function.

3) Generation of unique code sequences

Any code sequence possible at the assembler level should be possible to generate as a result of a procedure call with or without controlling parameters.

4) Addressing capabilities

The addressing conventions used by the processor also when running in special modes should be possible to handle as ADDRESS objects, and also be reflected in the definition of different POINTER types. Allocation of at least basic variable objects in any of the existing address spaces should be supported.

5) Process and interrupt control

In addition to the basic process and interrupt control in Modula-2 also hardware supported process control mechanisms should be possible to implement. Internal and external interrupt facilities should be possible to fully utilize and control.

1.3.2 Additional demands

The specifications of machine-dependent objects should be easy to define, change and delete. Even if a basic understanding of the implementation must be expected from anybody defining machine code sequences at this level, the implementing details should be hidden as much as possible.

Specifications associated with different components should be kept separate, i.e. in different modules. Such component modules should preferably be thought of as low-level library modules.

It should be possible to specify which low-level objects from different component modules should be included or omitted when a compiler for a new configuration is to be adapted.

The notation used for specification purposes should agree with the design philosophy of Modula-2. Special characters and complicated specification rules should be avoided.

With the specifications for a new configuration present the generation of the corresponding adapted compiler should be 'automatic', i.e. the generation should require no further manual intervention.

```
-----  
! The basic definition of Modula-2 must not be violated in !  
! any respect. The low-level definitions should be included !  
! in the pseudo module SYSTEM intended for that purpose. !  
-----
```

2.1 PRESENTATION

2.1.1 The basic idea

The basic idea behind the adaptable compiler system (ACS) is to provide the user (= systems programmer) with means to describe to the compiler how to implement operand types, operators and special code sequences for the components in a target configuration. The descriptions will be given in component modules in a notation close to Modula-2. The intention is that a systems programmer with basic knowledge of the implementation model should be able "program" the compiler with the aid of such formal descriptions. The component modules may also serve as pre-programmed library modules to be used when the Modula-2 compiler is adapted for more or less standard component configurations.

Given a set of component modules together with an implementor module containing the fundamental parts of a code generator for the target processor family, a new code generator part of the Modula-2 compiler will be generated together with configuration information for the fix front end part.

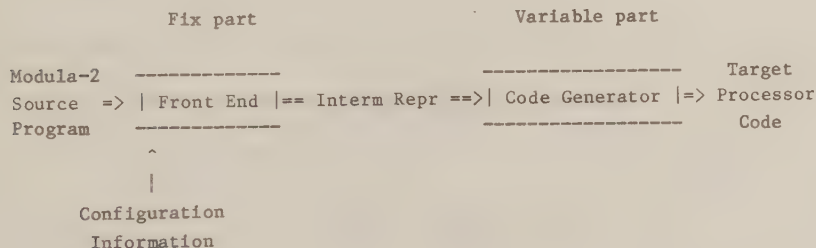


Fig. 2.1
The resulting Modula-2 compiler

Depending on the contents of the implementor module the program generated by the resulting Modula-2 code generator will be of stand-alone type or intended for execution under control of an operating system.

The ACS consists of three description module processors together with the fix front end part of the Modula-2 compiler. The basic tool used to construct each of the parts of the compiler system is the BOBS parser generator [Erik82] that will be briefly described before the presentation of the parts of the adaptable compiler system itself.

2.1.2 The BOBS parser generator

The BOBS-System is a parser generating system developed at the Computer Science Department of Aarhus University, Denmark. The system consists of a parser generator and a skeleton parser containing a general LALR(1) parser. (LALR(1) = Look Ahead Left-to-right scan Rightmost derivation parser examining 1 input symbol on each move). The parser generator will take as input a grammar and the source code of an input parser consisting of a skeletal parser and user supplied semantic routines.

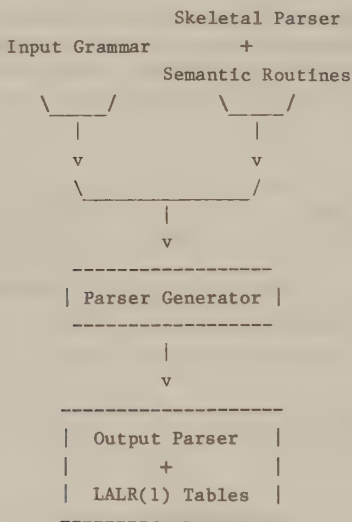


Fig 2.2
The BOBS Parser Generating System

The skeletal parser part contains the LALR(1) parser and basic support routines. The input grammar should contain a grammatical description

of the language to be parsed. The grammar will be thoroughly checked and possibly transformed by the parser generator before the corresponding LALR(1) tables are generated. The parser generator will add actual constant values to the input parser and produce an output parser tuned to the size of the input grammar.

The Input Grammar

The input grammar accepted by the BOBS system contains declarations and production rules. The declarations that must precede the grammar rules should specify the following items.

- Meta characters (defaulted if omitted)
- Terminal symbols (all terminals must be declared)
- String character (optional)
- Goalsymbol (first metasymbol if omitted)
- Comment symbols (optional)

The meta characters are used to express production rules in a notation corresponding to BNF.

Example

```
"Statement" ::= WHILE "Expr" DO "StatementSeq" END ?  
"VarList" ::= "VarList" , "Var" | "Var" ?
```

In the example above the meta characters are:

```
"    denoting the beginning and end of a metasymbol  
::= separating the left and right side of a production  
|    indicating alternatives  
?    terminating a production
```

The terminal symbols in the example are: WHILE DO END ,

Identifiers and numeric values are indicated by the symbols NAME and KONST respectively. The empty production is denoted by the symbol EMPTY. The symbol ERROR is used to indicate error actions.

A number of options (switches) are available for the control of printing, tests, internal representations and other built-in facilities.

2.1.3 The description part

The description part of the compiler system performs the processing of the description modules, i.e. the component modules, implementor modules, and the configuration modules.

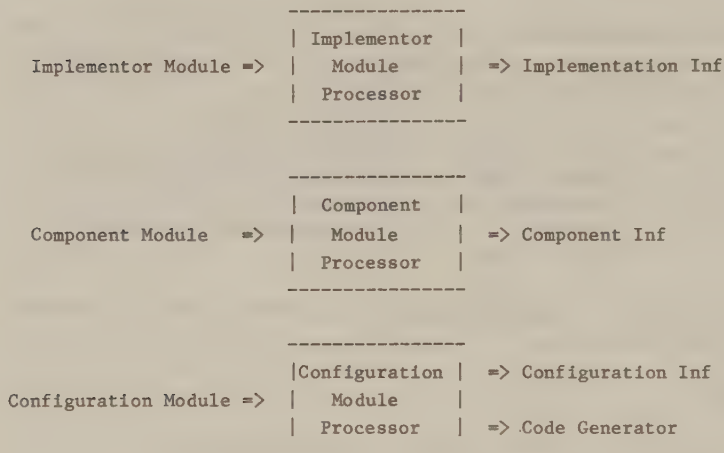


Fig 2.3
The Description Modules

The component modules contain specifications of types, operators and procedures specific for each component, such as a main processor, a coprocessor, an interrupt controller or a firmware module.

The implementor module represents a more hidden level of the implementation for a certain processor family. It contains implementation specific code rules and procedures supporting all the component modules included for the actual target architecture.

The configuration module specifies which objects defined in the component modules that should be included in the description of the configuration, i.e. the configuration information file read by the front end and the configuration rules and configuration code used when the new code generator is generated. The processing of the configuration module will also cause the corresponding code generator to be generated.

In the adaptable compiler system the BOBS parser generator is used to create parsers for the different parts of the system; the description modules, the compiler front end, and the code generator. In the first two cases it is chosen simply as a convenient tool while in the code generator case it plays a more fundamental role.

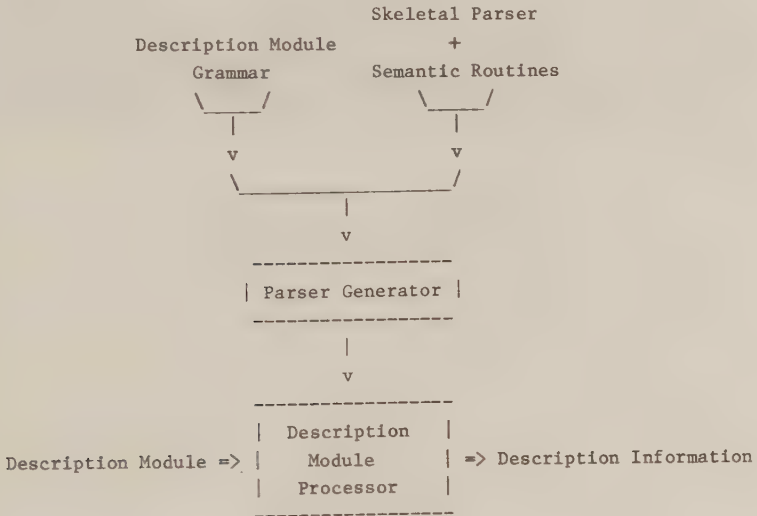


Fig. 2.4
An ACS Description Processor Generation

The implementor and component module processors are similar in many respects. Compared to the configuration module processor they are also more complex.

The description modules and the description information generated will be further discussed in chapter 5.

2.1.4 The front end part

The front end part of the compiler performs the lexical, syntactic and semantic analysis of a Modula-2 program. The input to the front end is a Modula-2 module and a description file generated by the description

processing part of the ACS. The output is the corresponding intermediate representation and a symbol table containing information concerning exported objects, if any.

The generation of the front end will require a grammar defining the standard Modula-2 syntax and the semantic routines for symbol table handling, storage allocation, type checking and intermediate code generation. The front end is generated once and may be used unaltered in Modula-2 compiler systems for different processor families. It differs from a conventional front end only in the respect that the information concerning standard objects and importable system objects is not built-in but collected from a "configuration information file" each time the Modula-2 compiler is invoked.

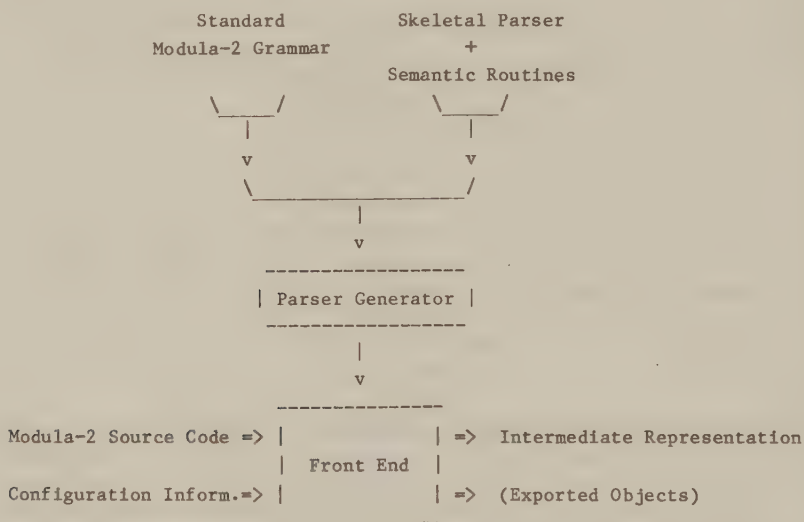


Fig. 2.5
The Front End Part

The Modula-2 source code may be a main module, a definition module, or an implementation module. The compiler will ask for the configuration module that prepared the actual configuration information file, or it will use a default name depending on the option set. The result of the front end processing is a sequence of Modula-2 statements in an intermediate representation that will be presented in chapter 6. In case the Modula-2 source code is a definition module a file containing information concerning exported objects will be generated.

2.1.5 The code generating part

The code generating part of the compiler system performs the synthesizing phases of the Modula-2 compiler. It takes as input the intermediate representation and will generate the corresponding code for the target processor.

The generation of the code generator part will require a grammar for the processing of the intermediate code together with semantic routines. The input grammar as well as the semantic routines will be composed by a fix part together with grammar rules and code sequences resulting from the processing of a configuration module.

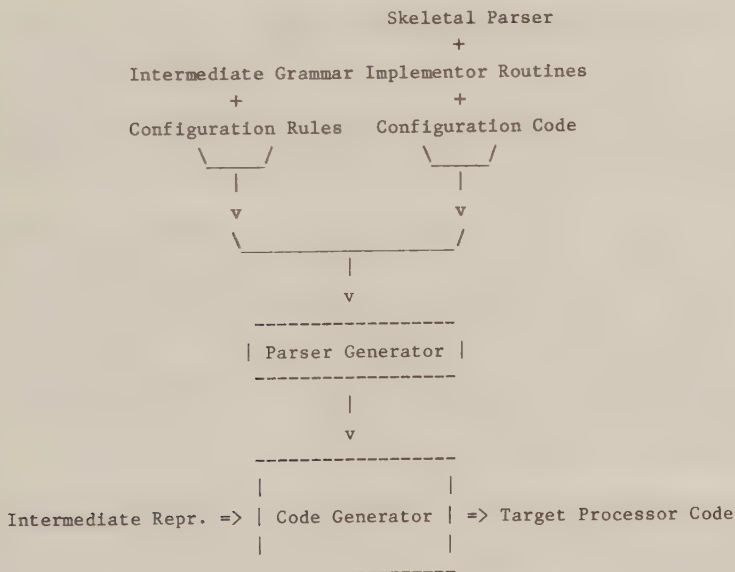


Fig. 2.6
The Code Generating Part

The fixed part of the grammar rules and the semantic routines concerns the operand addressing, control statement structures, procedure calls and parameter transfers. The configuration dependent part of the grammar rules and the semantic routines will define the processing of standard/system operators and procedures.

2.2 RELATED WORK

Lately, much interest has been directed towards the design of code generator writing systems [Catt78, Glan78, Grah80, Gana80, Kozl81]. The renewed interest is due to the promising results of applying pattern matching table-driven methods to the synthesizing back end part of the compiler in correspondence to the established use of such methods in the analyzing front end part. Most of this work has been aimed at the problem of retargetable code generation, i.e. given a formalized description of the target machine the code generator will be automatically produced. A comprehensive study of code generator writing systems is given in a thesis of Lunell [Lune83].

2.2.1 Table-driven code generation techniques

In the table-driven pattern matching code generation techniques the intermediate representation (IR) from the front end is compared with subpatterns to find a match. In many cases the IR is tree-based allowing efficient recursive tree-traversing algorithms to be applied.

Cattel [Catt78, Catt79, Catt80] uses a target machine independent IR called TCOL that is an attempt to create a universal tree representation for programs. The translation from the TCOL subtrees is specified by schemas describing the conditions for the match and the resulting code sequence to be generated. The goal of Cattel's work is to formalize the machine description and to solve the code selection problem for a complete programming language. Cattel is also concerned about generating optimal code sequences and he uses heuristic search algorithms for this purpose.

Glanville and Graham [Glan78, Grah80] chose a more low-level IR in the form of prefix notation. The tree-traversing relies on algorithms derived from the LR parsing technique with heuristics added to solve ambiguities in the input grammar. The major goal of this work was to develop a method for code selection suitable for retargeting of the code generator part of the compiler. The result of this approach was something of a breakthrough. First, it is possible to prove fundamental properties of the code generation algorithm; it will not block or loop given a valid input, and the code generated will be correct for a valid IR. Second, the resulting code generator is very efficient and generates code comparable with that of non-optimizing production compilers.

Ganaphati [Gana80,Gana82] extends the approach of Graham and Glanville with the use of semantic attributes and disambiguating predicates. The target machine is described by an attributed grammar [Knut68] instead of a context-free grammar [Aho73]. The code generation is performed by attributed parsing in a single bottom-up pass. Compared to Graham-Glanville this model is more structured and also more flexible. The grammar describing the target machine is composed by three classes of productions; addressing mode productions specifying the addressing scheme of the target machine, instruction selection productions specifying operation codes and restrictions for every machine data type, and transfer productions specifying the conversion of a machine data type to every other data type. The conversion to correct storage types are ensured by predicates and semantic attributes.

The following rule from [Gana82] describes three forms of adding byte operands in the VAX-11/780.

```
Byte<r  ->  + Byte<a Byte<r IsOne(>a) NotBusy(>r)
              EMIT(>`incb` >r)

          + Byte<r Byte<a IsOne(>a) NotBusy(>r)
              EMIT(>`incb` >r)

          + Byte<a.Byte<r TwoOp(>+>a>r)
              EMIT(>`addb2` >a >r)

          + Byte<r Byte<a TwoOp(>+>a>r)
              EMIT(>`addb2` >a >r)

          + Byte<a Byte<b GETTEMP(>`byte` <r)
              EMIT(>`addb3` >a>b>r)
```

The notation above uses >x and <x to express inherited and synthetic attributes respectively.

The disambiguating predicates are: IsOne(),NotBusy() and TwoOp().

The words in capital letters are "action symbols", i.e. calls to support procedures. In the example two action symbols are recognized: EMIT() and GETTEMP().

2.2.2 Similarities and differences

The problem faced in the adaptable compiler system has obvious similarities with the retargetability problem. However, the objectives are somewhat different. The research in the area of retargetable code generation techniques has primarily had the following goals.

- obtaining machine independence in the code generator
- efficient and systematic code selection
- formalization of machine descriptions
- development of language- and machine-independent intermediate representations

In the adaptable compiler system the code generator has not to be machine independent. The basic support procedures in the implementor module should rather be redesigned for each microprocessor family.

Efficient code selection will neither be a primary goal in the adaptable compiler system. However, the approach will not exclude the possibility to find optimal code sequences. The same is valid for efficient register allocation.

The description of the target architecture is not formalized in the sense that available machine instructions, addressing modes and register sets are specified in a special machine description section. The code generating sequences are specified as templates programmed in Modula-2 describing the implementation of the intermediate representation in the target processor including the utilization of the registers.

The intermediate representation generated by the front-end is designed specifically for Modula-2. It is not the intention that it should be a convenient representation also for other languages.

The objective of the adaptable compiler system is to provide a programmable extension to the high-level language Modula-2 adapted to the requirements of the application and target configuration.

The problems concern

- the syntactic definitions of the specificable items
- the organization of the description part
- the adjustment to the BOBS requirements
- the interface to Modula-2

Comparisons to Ganapathi's approach

Compared to the generation techniques briefly described in section 2.2.1 the approach taken in the adaptable compiler system has most in common with that of Ganapathi.

The main similarities are

The intermediate language is described by an attribute grammar and the code generation is realized by attribute parsing.

The code generation is performed in a single bottom-up pass.

The information contained in the IR is accumulated in attributes attached to the non-terminals during the parsing and used to guide the code emitting alternatives.

Code templates are attached directly to the grammar rules.

The main differences are

1. Disambiguating predicates are not used in the ACS

The purpose of the disambiguating predicates in Ganapathi's descriptions is twofold. First, they will handle pattern-matching conflicts during the parsing. Second, they will express the semantic condition that must be valid if a certain grammar rule is to be applied. In the ACS grammar conflicts can not occur as the BOBS system will assure this. The second role of the disambiguating predicates in the Ganapathi model is expressed by conditional Modula-2 statements in the ACS. This means that in the ACS the grammar rule is selected first and the conditions evaluated thereafter. In most cases the approaches are equivalent, but there might be situations where the direct control by predicates of the reduction sequence is more advantageous. On the other hand the disambiguating predicates may block the parsing if not used properly.

2. Inherited attributes are not used in the ACS

The lack of inherited attributes due to the limitations of the BOBS system must be compensated by central tables containing semantic information otherwise passed down the parse tree. This

is no real disadvantage as the use of inherited attributes tend to make the specifications difficult to construct and to understand.

3. The code templates are not restricted to action symbols.

In the ACS general Modula-2 statements are used in the code templates. Compared to the action symbols, i.e. procedure calls in Ganapathi's scheme, the statement approach is more expressive and also necessary in order to specify the alternatives corresponding to the disambiguating predicates. References within a code template are restricted to semantic attributes and implementation procedures only.

2.2.3 Notational conventions

The definitions of the syntactic constructs in the description modules will use the Extended Backus-Naur Formalism (EBNF) in agreement with the definitions in the Modula-2 Report [Wir82]. The original Modula-2 syntax rules will accordingly be marked with a "\$" in the left marginal, while the syntax rules introduced for the specifications in the compiler system will be marked with a "\$+" in the left marginal.

Example

```
$+  ComponentModule = COMPONENT MODULE ident ";"
$+      [export]
$+      {import}
$+      {ComponentSpec}
$+      END ident "."
```

[...] square bracket indicates that the enclosed sequence of symbols is optional.

{...} braces indicate that the enclosed symbols may be repeated zero or more times.

CHAPTER 3. IMPLEMENTATION-DEPENDENT ASPECTS OF MODULA-2

In this chapter the implementation-dependent aspects of Modula-2 will be summarized. The presentation will form a basis for the definitions of implementation-dependent specifications in the next chapter. The definitions in this chapter are according to "Programming in Modula-2" [Wirt82].

3.1 STANDARD OBJECTS

3.1.1 Standard data types

The basic standard data types in Modula-2 are

- a) The type `INTEGER` representing integer numbers between implementation-dependent limits.
- b) The type `CARDINAL` representing non-negative whole numbers starting with 0 up to an implementation-dependent upper limit.
- c) The enumeration type `BOOLEAN` representing the logical truth values denoted by the standard identifiers `TRUE` and `FALSE`.
- d) The type `CHAR` representing an implementation-dependent character set.
- e) The type `REAL` representing real numbers using an implementation-dependent floating-point format.
- f) The type `SET` representing the set of all possible sets consisting of elements from a given base type. The maximum number of elements in the base type is implementation dependent. Especially there is the pre-defined type `BITSET` representing the set of integers between 0 and $N-1$, where N is an implementation-dependent constant equal to the wordlength of the processor, or a small multiple of it.
- g) The enumeration type representing a list of values denoted by constant identifiers, which belong to this type only. The maximum set of enumerated values is implementation dependent.

The Subrange type

A subrange type represents a limited, but contiguous set of values belonging to a certain base type. The implementation of the subrange type is dependent on that base type.

```
$ SubrangeType = "[" ConstExpression ".." ConstExpression "]"
```

Especially the implementation must support run-time checks of the value assignments to objects of subrange type.

The Array type

Array structured data types consists of a collection of elements of the same type, that each may be accessed by a selecting index.

```
$ ArrayType = ARRAY SimpleType {"", SimpleType} OF type.
```

Arrays in Modula-2 are declared with a fixed number of elements. However, a formal parameter to a procedure may be declared as an "open array" leaving the index range open.

```
$ FormalType = ["ARRAY OF"] qualident.
```

The implementation dependency concerns the following points:

- 1) Run-time checks of the index values.
- 2) Addressing of elements in single and multi-dimensional arrays.
- 3) Word alignment of the elements depending on the requirement of the processor.

The Record type

The record structure contains a collection of possibly different elements declared as a unit. In addition, the record type may contain

so called variant parts specifying elements allocated at shared memory locations.

```
$ RecordType = "RECORD"  
$           FieldListSequence  
$           "END".
```

The implementation of record type concerns the following points:

- 1) Run-time checks of the tag-field in the variant parts.
- 2) Addressing of the elements.
- 3) Word alignment of the elements if required.

The Pointer type

Variables of a pointer type P assume as values pointers to variables of another type T. The pointer type P is said to be bound to T, i.e. variables of pointer type P can point only to objects of type T.

```
$ PointerType = POINTER TO type.
```

The representation of the pointer value (as an address) is implementation-dependent.

A new pointer value is generated by a call to the standard procedure NEW.

Example

```
TYPE link = POINTER TO element;  
VAR l: , link;  
...  
NEW(l);  
...
```

The created variable is dynamically allocated, it has no name, is anonymous, and can be accessed only via pointers using the dereference operator "^". The value NIL indicates that a pointer variable is pointing to no object.

Deletion of a dynamically allocated variable is performed by the standard procedure DISPOSE that frees the storage space allocated to the variable pointed at by 1.

Example

```
DISPOSE(1);
```

The statements NEW(1) and DISPOSE(1) are translated by the compiler into

```
ALLOCATE(1,TSIZE(link)) and  
DEALLOCATE(1,TSIZE(link))
```

A program using NEW or DISPOSE must either import ALLOCATE and DEALLOCATE or explicitly declare them. The TSIZE system procedure denotes the number of storage units assigned to the type specified as a parameter.

As a consequence of defining the ALLOCATE and DEALLOCATE procedures at the program level their implementation is of no concern to the compiler.

The Procedure type

Modula-2 regards a procedure declaration as a special kind of a constant declaration, the value of this constant being a procedure. Variables declared as procedure types may be assigned values of procedures that are compatible with their declaration.

```
$ ProcedureType = "PROCEDURE"[FormalTypeList].
```

A call to a variable of procedure type will cause the procedure assigned to it to be executed in the same way as if it was called by a direct reference. Procedures assigned to variables of procedure type must be declared at the outermost level. Standard procedures are not assignable.

The representation of the procedure type will depend on the procedure calling instructions of the actual processor, as well on the overall organization of an executable Modula-2 program in a specific implementation.

3.1.2 Standard operators

Arithmetic operators

The operations are applicable also on operands belonging to a subrange of the specified type, with exception for REAL.

Symbol -----	Operation -----	Operand Types -----	Result Type -----
+	identity	INTEGER	INTEGER
		CARDINAL	CARDINAL
		REAL	REAL
-	sign inversion	INTEGER	INTEGER
		REAL	REAL
+	addition	INTEGER, INTEGER	INTEGER
		CARDINAL, CARDINAL	CARDINAL
		REAL, REAL	REAL
-	subtraction	INTEGER, INTEGER	INTEGER
		CARDINAL, CARDINAL	CARDINAL
		REAL, REAL	REAL
*	multiplication	INTEGER, INTEGER	INTEGER
		CARDINAL, CARDINAL	CARDINAL
		REAL, REAL	REAL
/	real division	REAL, REAL	REAL
DIV	integer division	INTEGER, INTEGER	INTEGER
		CARDINAL, CARDINAL	CARDINAL
MOD	modulus	INTEGER, INTEGER	INTEGER
		CARDINAL, CARDINAL	CARDINAL

The table clearly indicates that the operands of an arithmetic operation must be of the same type, i.e. Modula-2 does not support implicit type conversion in arithmetic operations. The only way to mix operands of different types in an arithmetic operation is to explicitly apply a type transfer function to either of the operands (see 3.3). This is true also for other dyadic Modula-2 operators with the only exception for the assignment operator where CARDINAL and

INTEGER types are considered compatible.

Logical Operators

Symbol -----	Operation -----	Operands -----	Result -----
NOT	negation	BOOLEAN	BOOLEAN
OR	conjunction	BOOLEAN, BOOLEAN	BOOLEAN
AND, &	disjunction	BOOLEAN, BOOLEAN	BOOLEAN

Modula-2 states that "short circuit" evaluation is applied for the OR and AND operations, i.e. the second operand is not evaluated if the result of the operation is already known by the evaluation of the first operand.

Set Operators

Symbol -----	Operation -----	Operands -----	Result -----
+	set union	SET, SET	SET
-	set difference	SET, SET	SET
*	set intersection	SET, SET	SET
/	symmetric set difference	SET, SET	SET

In term of set membership the set operations are defined as:

set union	$i \text{ IN } (u+v)$	iff	$(i \text{ IN } u) \text{ OR } (i \text{ IN } v)$
set difference	$i \text{ IN } (u-v)$	iff	$(i \text{ IN } u) \text{ AND NOT } (i \text{ IN } v)$
set intersection	$i \text{ IN } (u*v)$	iff	$(i \text{ IN } u) \text{ AND } (i \text{ IN } v)$
symmetric set difference	$i \text{ IN } (u/v)$	iff	$((i \text{ IN } u) \text{ <> } (i \text{ IN } v))$

(i denotes a set element, u and v denote sets)

Relational Operators

In the table below the operand type "Opnd" denotes one of the basic types INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, enumeration, or an operand belonging to a subrange of one of these (with exception for REAL).

Symbol -----	Operation -----	Operands -----	Result -----
=	equal	Opnd,Opnd	BOOLEAN
		POINTER,POINTER	BOOLEAN
		SET,SET	BOOLEAN
#,<>	unequal	Opnd,Opnd	BOOLEAN
		POINTER,POINTER	BOOLEAN
		SET,SET	BOOLEAN
<	less	Opnd,Opnd	BOOLEAN
<=	less or equal	Opnd,Opnd	BOOLEAN
<=	proper set inclusion	SET,SET	BOOLEAN
>	greater	Opnd,Opnd	BOOLEAN
>=	greater or equal	Opnd,Opnd	BOOLEAN
>=	improper set inclusion	SET,SET	BOOLEAN
IN	set membership	SetElement,SET	BOOLEAN

Comparison between operands of structured type, i.e. records and array types, are not allowed.

Assignment Operator

Operands are assignment compatible, if they are either of the same type or, if both are of INTEGER or CARDINAL type or subranges with base types of INTEGER or CARDINAL.

Symbol -----	Operation -----	Operands -----	Result -----
:=	assignment	INTEGER,INTEGER	INTEGER
		CARDINAL,CARDINAL	CARDINAL
		BOOLEAN,BOOLEAN	BOOLEAN
		CHAR,CHAR	CHAR
		REAL,REAL	REAL
		SCALAR,SCALAR	SCALAR
		SET,SET	SET
		INTEGER,CARDINAL	INTEGER
		CARDINAL,INTEGER	CARDINAL

Structured types are assignment compatible only if they are explicitly declared to be of the same type.

Operator precedence

The unary operators have the highest precedence.

Unary Operators: + | - | NOT

Of the binary operators the so-called multiplying operators have the highest precedence, followed by the adding operators, the relational operators and, with lowest priority, the assignment operator. Parentheses may be used to change the evaluation order.

Multiplying Operators: * | / | DIV | MOD | AND | &

Adding Operators: + | - | OR

Relational Operators: = | # | <> | < | <= | > | >= | IN

Assignment Operator: :=

A sequence of operators of the same priority is by definition executed from left to right.

3.1.3 Standard procedures

The predefined standard procedures expected to be present in any implementation are the following.

ABS(x)	absolute value; result type = argument type.
CAP(ch)	if ch is a lower case letter, the corresponding capital letter; if ch is a capital letter, the same letter.
CHR(x)	the character with ordinal number x. CHR(x) = VAL(CHAR,x)
FLOAT(x)	x of type CARDINAL represented as a value of type REAL.
HIGH(a)	high index bound of array a.
ODD(x)	$x \text{ MOD } 2 \neq 0$.
ORD(x)	ordinal number (of type CARDINAL) of x in the set of values defined by the type of x. T is any enumeration type, CHAR, INTEGER, or CARDINAL.
TRUNC(x)	real number x truncated to its integral part (of type CARDINAL).
VAL(T,x)	the value with ordinal number x and with type T. T is any enumeration type, CHAR, INTEGER, or CARDINAL. VAL(T,ORD(x)) = x, of x of type T.
DEC(x)	$x := x - 1$
DEC(x,n)	$x := x - n$
EXCL(s,i)	$s := s - \{i\}$
HALT	terminate program execution
INC(x)	$x := x + 1$
INC(x,n)	$x := x + n$
INCL(s,i)	$s := s + \{i\}$

The procedures INC and DEC also apply to operands of enumeration types and of type CHAR. In these cases they replace x by its (n-th) successor or predecessor.

3.2 SYSTEM OBJECTS

System objects are those low-level objects that are highly implementation dependent. Modula-2 programs using such objects are not immediately portable to other Modula-2 systems.

3.2.1 System data types

The basic system data types in Modula-2 are

- a) The type WORD denoting an individually accessible storage unit with a fixed number of bits. No operators other than assignment are applicable to data of type WORD.

If a formal parameter in a procedure is of type WORD the actual parameter may be of any type that is allocated in a corresponding storage unit.

If a formal parameter has the type ARRAY OF WORD then its actual parameters may be of any type.

- b) The type ADDRESS defined as

ADDRESS = POINTER TO WORD

The ADDRESS type is compatible to all pointer types and also to the type CARDINAL. Therefore, all operators for cardinal arithmetics also applies to the ADDRESS type.

The process type

A process represents a concurrent execution path in a program. Processes are executed quasi-concurrently by transfer statements programmed at the Modula-2 level, i.e. the Modula-2 concept of processes is rather that of coroutines.

The PROCESS type is implemented by a package that at least should allocate space for a reference to a working space (stack) intended to contain status information and procedure frames. The size and structure of the PROCESS type will depend on the addressing characteristics of the processor and the organization of an executing Modula-2 program.

3.2.2 System procedures

The basic system procedures are:

The function `ADR(VAR x: AnyType)` denoting the address to the variable "x". The result of the `ADR` function is of type `CARDINAL`.

The function `SIZE(VAR x: AnyType)` denoting the number of storage units assigned to the variable "x". The result of the `SIZE` function is of type `CARDINAL`.

The function `TSIZE(AnyType)` denoting the number of storage units assigned to any variable of "AnyType". The result of the `TSIZE` function is of type `CARDINAL`.

The procedure

```
NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL; VAR new: PROCESS);
```

will create a workspace at address "A" of size "n" to be used by process "p". When started the process will execute procedure "P". The implementation will be discussed in chapter 9.

The procedure

```
TRANSFER(VAR p1,p2: PROCESS)
```

will suspend the current process and assign it to "p1", and then "p2" is resumed. The process "p2" must have been assigned a process by an earlier call to either `NEWPROCESS` or `TRANSFER`. The implementation will be discussed in chapter 9.

The procedure ,

```
IOTRANSFER(VAR p1,p2: PROCESS; va: CARDINAL)
```

in analogy to `TRANSFER` will suspend the current (device) process and assign it to "p1", and then "p2" will be resumed. In addition the next interrupt associated to vector entry "va" will cause the interrupted process to be assigned to "p2" and then "p1" will be resumed. The implementation will be discussed in chapter 10.

3.3 TYPE COMPATIBILITY

Modula-2 regards two type declarations as compatible only if they are explicitly declared to be equal.

Example

```
TYPE
    T1  =   ARRAY [0..15] OF INTEGER;
    T2  =   ARRAY [0..15] OF INTEGER;
    T3  =   T1;
```

In the example above, operands of type T3 are compatible to those of type T1. Operands of type T1 and T2 are not compatible in spite of the fact that the right hand sides of their declarations are identical.

Type transfer functions

The standard operations are strictly limited to operate on operands of the types specified in the tables in section 3.1.2. However, it is suggested that the implementation supports so-called "type transfer functions" that will convert an operand to another type.

Example

```
VAR i:   INTEGER;
    c:   CARDINAL;
    ...
    i:=i+INTEGER(c);
    c:=c+CARDINAL(i);
```

Type transfers takes place during compilation and should not cause any computation during run-time, indicating that the data types involved should have compatible lengths.

Type conversion functions

Type conversion functions, on the other hand, will alter the representation of an operand during run-time. Modula-2 has two such predefined functions; FLOAT and TRUNC. FLOAT(c) will convert the CARDINAL operand "c" to a REAL floating point operand, while TRUNC(r) will do the opposite. The implementation of type conversion functions should utilize the hardware support, i.e. numeric coprocessors rather

than separately coded routines. In some processors this will imply that the INTEGER type rather than the CARDINAL type will be involved in the conversion.

3.4 THE SYSTEM MODULE

In order to isolate machine-dependencies Modula-2 introduces the pseudo-module SYSTEM. The SYSTEM module is intended to contain data types and procedures strongly dependent of the actual implementation. Any module importing objects from the SYSTEM module will be considered as a low-level module needing careful examination if transferred to another Modula-2 implementation. The SYSTEM module is not a module in the normal sense but can be considered as being an integrated part of the compiler and is therefore called a pseudo-module.

However, if defined by a definition module the specification of the SYSTEM module would look like:

DEFINITION MODULE SYSTEM;

EXPORT QUALIFIED

WORD, ADDRESS, PROCESS,

ADR, SIZE, TSIZE,

NEWPROCESS, TRANSFER, IOTRANSFER, . . .;

TYPE WORD; ADDRESS, PROCESS,

PROCEDURE ADR(a: AnyType): ADDRESS;

PROCEDURE SIZE(v: AnyType): CARDINAL;

PROCEDURE TSIZE(AnyType): CARDINAL;

PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL;

VAR q: PROCESS);

PROCEDURE TRANSFER(VAR p,q: PROCESS);

PROCEDURE IOTRANSFER(VAR p,q: PROCESS; va: CARDINAL);

. . .

END SYSTEM.

The dots (. . .) imply that the SYSTEM module may contain additional facilities depending on the actual implementation. This intention is fundamental for the adaptable compiler system, as the SYSTEM module will serve as the software interface needed to introduce non-standard component and configuration dependent data types, operations and procedures.

CHAPTER 4. COMPONENT MODULE SPECIFICATIONS

The purpose of the specifications in a component module is to define those implementation dependent issues that are of primary concern for the programmer to control and utilize in the corresponding hardware component. The adaptable compiler system will expect at least one component module defining the processor characteristics when a new Modula-2 compiler is to be generated.

The specifications within a component module will be classified as:

- I) Standard/System Types
- II) Standard/System Operators
- III) Standard/System Procedures
- IV) Space Allocation Prefixes

The specifications should be prefixed by the keywords STANDARD or SYSTEM in order to make a separation between the implementation control of the standard Modula-2 objects and those that will be included in the pseudo-module SYSTEM. The standard specified objects will be valid for reference in all parts of a Modula-2 program, while the system specified objects will be valid in a Modula-2 module only if they are explicitly imported from the SYSTEM module.

The syntactic definitions in this and subsequent chapters should not be confused with the original Modula-2 syntax. To make the distinction quite clear the syntactic definitions used in connection with the ACS will be marked with a "\$+" in the left marginal, while the original Modula-2 definitions in the last chapter were marked with a "\$" only.

The basic syntax for the component module is

```
$+ ComponentModule = COMPONENT MODULE ident ";"
$+                 [export]
$+                 {import}
$+                 {ComponentSpec}
$+                 END ident "." .

$+ ComponentSpec = StandardTypeSpec |
$+                 SystemTypeSpec |
$+                 StandardOpDecl |
$+                 SystemOpDecl |
$+                 StandardProcDecl |
$+                 SystemProcDecl |
$+                 SpacePrefixSpec.
```

The export list specifies the objects possible to include in the compiler generation for a certain configuration, i.e. they will be available for import into a configuration module. The import list is supposed to import support objects from the associated implementor module (see next chapter).

4.1 TYPE SPECIFICATIONS

4.1.1 Standard type specifications

A standard type specification states that a basic Modula-2 type will be valid for reference at the program level and in subsequent specifications within the component module itself. The type specification also defines the allocation size of the data type expressed in bits.

```
$+ StandardTypeSpec = STANDARD TYPE StandardTypeList.  
$+ StandardTypeList = StandardType {";" StandardType}.  
$+ StandardType = ident "=" BasicType [OF] number.
```

The "BasicType" must be one of the standard type identifiers; INTEGER, CARDINAL, BOOLEAN, CHAR, REAL, SET, enumeration, POINTER or PROCEDURE. For enumeration types the SCALAR identifier is used. All characteristics of the basic type at the right side of the specification will be preserved by the type on the right hand side, with the exception for the allocation size that is specified by the "number" part.

Example

STANDARD.TYPE

```
INTEGER    =    INTEGER OF 16;  
CARDINAL   =    CARDINAL OF 16;  
BOOLEAN    =    BOOLEAN OF 8;  
CHAR       =    CHAR OF 8;  
BITSET     =    BITSET OF 16;  
SET        =    SET OF 64;  
SCALAR     =    SCALAR OF 16;  
POINTER    =    POINTER OF 32;  
PROCEDURE  =    PROCEDURE OF 32;
```


The component module analyzer will only allow standard Modula-2 type identifiers in standard type declarations. The type identifiers in the left and right hand side must be identical, i.e. a standard type cannot be redefined with another name. The declared standard type identifiers will be automatically imported into all modules at the program level. On the other hand, standard data types that have not been declared within any component module will not be valid for reference at the program level. For example the REAL standard type could be left undefined in configurations not supporting floating point operations.

4.1.2 System type specifications

The syntax for the specification of system types is

```
$+ SystemTypeSpec = SYSTEM TYPE SystemTypeList.
$+ SystemTypeList = SystemType {";" SystemType}.
$+ SystemType = ident "=" BasicType [OF] number |
$+             ident "=" PointerType TO SystemType |
$+             ident "=" BitStrucType.
```

The "BasicType" includes in this case the basic system type identifiers WORD or PROCESS in addition to the basic standard type identifiers. The left side identifier does not have to be identical to the right hand side basic type identifier. The characteristic of the right side type will be inherited apart from the allocation size that is explicitly specified by the "number" part. New pointer types will also be permissible as system types. The "BitStrucType" denotes a bit structured type that will be described at the end of this section.

Multiple precision types

The type specifications provide a mean to specify basic standard types of any precision supported by the implementation. In this way two levels of basic types are established. The standard type specification for each basic type defines which precision should be generally used at the program level, while the corresponding system type specifications define the supplementary precisions. It is necessary to give such additional type specifications system status in order not to violate the single precision type concept in Modula-2 and to cause any module importing the additional types to be marked as an implementation-dependent low-level module. Of course the specification of a standard type is also implementation-dependent, but for a

different reason. Any implementation of Modula-2 must define the generally available basic types according to the precision supported by the hardware, but the possibility to define additional basic types is unique to this implementation and such types should consequently be isolated in the SYSTEM module.

Examples

SYSTEM TYPE

```
SHORTINTEGER  =  INTEGER OF 8;
LONGINTEGER   =  INTEGER OF 32;

SHORTSET      =  SET OF 8;
LONGSET       =  SET OF 256;

SHORTPOINTER  =  POINTER OF 16;
LONGPOINTER   =  POINTER OF 32;
```

As pointed out previously, each specified type must be supported with the specifications of the associated operators. The operator specifications are discussed in section 4.2 .

Word types

The WORD system type is used to specify storage types of desired size. No operation except assignment is allowed for word types. If a formal parameter of a procedure is specified to be of a word type any actual type of the same allocation size will be considered compatible.

Examples

```
SYSTEM TYPE BYTE      =  WORD OF 8;
SYSTEM TYPE WORD       =  WORD OF 16;
SYSTEM TYPE DOUBLEWORD =  WORD OF 32;
```

Address types

System types specified as pointers to storage unit types are considered as address types. They will inherit the allocation size of the pointer type used at the right side of the specification and will be assignment compatible with all pointer types of that size.

Examples

SYSTEM TYPE

BYTEOFFSET	=	SHORTPOINTER TO BYTE;
WORDOFFSET	=	SHORTPOINTER TO WORD;
BYTEADDRESS	=	LONGPOINTER TO BYTE;
WORDADDRESS	=	LONGPOINTER TO WORD;

In [Wirt82] it is recommended that arithmetic cardinal operations should be applicable to the address type. This will not be automatically true in a Modula-2 compiler generated by this system. Rather the implementor is required to define all operations, except for the assignment, for each specified address type. The reason for this is that the format for addresses is very machine specific:

iAPX86: 16 bit "segment paragraph address" right shifted 4 bits, plus a 16 bit offset address
Z8000: 7 bit "segment number" right justified in the most significant byte of a 16 bit segment selector, plus a 16 bit offset address
MC68000: 24 bit "long addresses"
NS16000: 32 bit "pointer addresses"

Process system types

The realization of the Modula-2 processes (coroutines) depends on the overall organization of the executable Modula-2 program, and of the hardware support for process transfers including interrupt responses. However, each process requires a workspace where activation records and process status information will be kept. A process object must therefore include a reference to that workspace. In most implementations a process variable must allocate space for that reference only, in others with hardware supported processes a packet containing additional information may be needed.

Example

SYSTEM TYPE PROCESS	=	PROCESS OF 16
SYSTEM TYPE TASK	=	PROCESS OF 128

The further details of process implementation will be discussed in chapter 9.

Bit structured system types

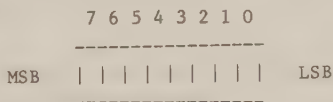
The definition of Modula-2 does not allow a bit field within a machine word to be referenced or assigned a value in any other way than by the aid of set operations. Manipulation of bit fields is very frequent when programming at low-level in a micro system environment. It is important that such manipulations can be expressed in a clear way and also it is desirable that operations on bit fields are efficiently implemented. Modula-2 has a concern for simplicity that should be praised, but for a systems programming language the lack of bit field support is a disadvantage.

The Modula-2 compiler generated by this system will allow a restricted form of bit structured types. In order not to violate the basic definition of Modula-2 the bit structured types will be possible to define only in system type specifications thus implying that any program module importing such a type will become an implementation dependent low-level module. The object itself may be passed as a parameter but not references to the bitfields.

The syntax of a bit structured type specification is

```
$+ BitStrucType   = PrefixType RECORD BitFieldList END.
$+ BitFieldList  = . BitField {";" BitField}.
$+ BitField      = [BitIdent ":" SimpleType].
$+ BitIdent      = ident "[" "BIT:" ConstExpression.
$+               [".." ConstExpression] "]".
```

The structured bit field record will be given the type of the prefix type that must be a previously defined standard or system type. The prefix type will also define the allocation size, i.e. the total number of bits in the bit field structure. The implementation may restrict this size to basic word length only. Single bits are referenced from right to left starting with 0. This reference order corresponds to the praxis in most manufacturers' component specifications. The elements within a bit field record may overlap each other. All bitfields need not necessarily be specified.



The type of a bitfield may not have a length exceeding the number of

bits in the bitfield, i.e. CARDINAL and INTEGER types should be specified as subranges in accordance with the bitfield lengths. A bitfield of enumeration type will require the size necessary to enumerate the elements. A bitfield of BOOLEAN type will consequently be considered to have the size of one bit.

Example

SYSTEM TYPE

InterruptType = BYTE RECORD

SpecificRotation [BIT: 7]: BOOLEAN;
AutomaticRotation[BIT: 6]: BOOLEAN;
EndOfInterrupt [BIT: 5]: BOOLEAN,
InterruptPriority[BIT: 4]: (Low,High);
InterruptLevel [BIT: 0..2]: [0..7];

END;

The bitfields are referenced and assigned values in the same way as ordinary record elements.

Example

FROM SYSTEM IMPORT InterruptType;

...

VAR a: InterruptType;

...

a.AutomaticRotation:=TRUE;

a.InterruptPriority:=High;

a.InterruptLevel:=3;

Another application for the bit structured types is user coded type conversion routines.

The bit structured types will require that bitfield information is included in the type information produced by the component module analyzer. They will also affect the addressing information in the intermediate representation.

4.2 OPERATOR SPECIFICATIONS

For each standard or system data type declared the associated set of Modula-2 operators for that data type is expected to be implemented. This is true also for data types of the same basic type having different allocation sizes, i.e. another precision.

The syntactic rule for the declaration of an operator in the component module is

```
$+ StandardOpDecl = STANDARD OpDecl.
```

```
$+ SystemOpDecl = SYSTEM OpDecl.
```

```
$+ OpDecl = OPERATOR "(" Type Operator Type ")" ":" Type ";"  
$+         [BEGIN  
$+             OpRuleSequence]  
$+         END.
```

The "Type" identifiers must refer to declared standard or system types, and the "Operator" should be a valid Modula-2 operator character. The "OpRuleSequence" specifies the grammar rules and the associated code templates for the possible combinations of operands. The number of rules within the operator declaration depends on the operand addressing capabilities of the target machine instructions and the implementation model used.

The following example defines the standard integer addition operator.

Example

```
STANDARD OPERATOR (INTEGER + INTEGER): INTEGER;  
BEGIN  
  RULE <Expr> ::= <Expr> + <Constant> DO  
    / Code template /  
  END;  
  
  RULE <Expr> ::= <Expr> + <Expr> DO  
    / Code template /  
  END;  
  
  RULE <Expr> ::= <Expr> + <Var> DO  
    / Code template /  
  END;  
END;
```

The SYSTEM prefix must be used whenever a system type is present in the specification.

Example

```
SYSTEM OPERATOR (BYTE := BYTE);
BEGIN
    RULE <Assignment> ::= <Var> := <Expr> DO
        / Code template /
    END;

    RULE <Assignment> ::= <Var> := <Constant> DO
        / Code template /
    END;
END;
```

The syntax for the rules in an operator specification is

```
$+ OpRuleSequence = OpRule {; OpRule}.
$+ OpCodeRule = RULE OpProduction [DO
$+             StatementSequence]
$+             END.
$+ OpProduction = NonTerminal "::~="
$+             [NonTerminal] Operator NonTerminal.
$+ NonTerminal = "<" ident ["." number] ">".
```

The non-terminal symbols used in the "OpProduction" should be defined in the basic grammar for the intermediate code. The non-terminal symbol attributes are defined by attribute declarations in the implementor module (see 5.1). The "Operator" must be the same as used in the operator specification head. The "number" indicator within the arrow brackets of a non-terminal symbol is used to distinguish between identical non-terminal symbols appearing more than once in the production. The first occurrence is referenced with 0.

Example

```
RULE <Assignment> ::= <Var.0> := <Var.1> DO

    EmitRM(  'MOV',    'BL', <Var.1> );
    EmitMR(  'MOV',    <Var.0>, 'BL' );
END;
```


In the last example the number notation is used to distinguish between the references to the attribute records of the $\langle \text{Var} \rangle$ non-terminal symbols. The "EmitRM" and "EmitMR" procedures will be called when the reduction is performed. They are code emitting procedures supposed to be defined in the implementor module (see 5.1). The operands of the code emitting procedures forms the operator and operands of the assembler instruction to be generated. The blanks between the operands are not significant but inserted for readability reasons.

If no reference to a non-terminal is made within the statement part, or if the non-terminal appears only once, no number indicator need to be used.

Example

```
RULE  $\langle \text{Expr} \rangle ::= \langle \text{Expr} \rangle + \langle \text{Expr} \rangle$  DO

    EmitR(    'POP',    'BX'    );
    EmitRR(   'ADD',    'AX', 'BX' );
END;
```

The production part of each rule will be included in the grammar for the intermediate representation after some modifications. For instance the "Operator" will be replaced by an internal operator that might be inserted into a prefix or postfix position.

Each non-terminal will have its own record of attribute elements associated, with the exception for the leftside non-terminal that will share the record with the first symbol on the right hand side of the production. Within the statement part the attributes of the non-terminals may be referenced as elements of this record, or the record as a whole may be referenced by the non-terminal itself.

Examples

```
RULE  $\langle \text{Expr} \rangle ::= \langle \text{Expr} \rangle + \langle \text{Constant} \rangle$  DO

    EmitRC(    'ADD',    'AX',  $\langle \text{Constant} \rangle$ .value    );
END;

RULE  $\langle \text{NDP} \rangle ::= \langle \text{NDP} \rangle + \langle \text{Constant} \rangle$  DO

    EmitNDPC(  'FADDP', Real32,     $\langle \text{Constant} \rangle$     );
END;
```

4.3 PROCEDURE SPECIFICATIONS

4.3.1 Standard procedure specifications

For each standard type all applicable standard procedures must be implemented.

The syntactic rule is

```
$+ StandardProcDecl = STANDARD PROCEDURE standardprocident
$+                      ["(" TypeParamList ")"]
$+                      [":" Type]]
$+                      [BEGIN
$+                      ProcRuleSequence]
$+                      END.

$+ TypeParamList = TypeParam {"," TypeParam}.

$+ TypeParam = [VAR | CONST] [ident ":" ] Type.

standardprocident = ABS | CAP | CHR | FLOAT | HIGH | ODD |
                   ORD | TRUNC | VAL | DEC | EXCL | HALT |
                   INC | INCL.
```

The component module processor, as in the case of operator specifications, will generate an internal operator that will stand for references to the standard procedure in the intermediate representation. The code rules are written in a similar style as the operator specifications, but the production part is slightly changed to reflect the procedure call.

```
$+ ProcRuleSequence = ProcCodeRule {";" ProcCodeRule}

$+ ProcCodeRule = RULE ProcProduction [DO
$+                      StatementSequence]
$+                      END.

$+ ProcProduction = NonTerminal ":@"
$+                      standardprocident ["(" NonTerminalList ")"].

$+ NonTerminalList = NonTerminal {"," NonTerminal}.
```

An example of the specification of the ABS standard procedure for INTEGER arguments is

Example

```

STANDARD PROCEDURE ABS (i: INTEGER): INTEGER;
BEGIN

    RULE <Expr> ::= ABS ( <Expr> ) DO

        EmitRI(  ^CMP^,    ^AX^,0          );
        EmitA(   ^JP^,    ^$+2^          );
        EmitR(   ^NEG^,    ^AX^          );
    END;

    RULE <Constant.0> ::= ABS ( <Constant.1> ) DO

        <Constant.0>:=ABS(<Constant.1>.value);
    END;
END;
```

The first rule makes use of code emitting procedures, i.e. "EmitRI", "EmitA" and "EmitR". The second code rule above will not emit any code as the result can be evaluated during compile time (using the ABS standard procedure in the compiling Modula-2 compiler).

It is up to the programmer to decide if the standard (or system) procedures should be realized in straight code, or if a call to a central routine should be generated when the reduction is applied. In the latter case the central routine must be present in the run-time system. The inclusion of run-time routines will be discussed in chapter 7.

The NEW and DISPOSE standard procedures

The standard procedures NEW and DISPOSE are translated into calls to the ALLOCATE and DEALLOCATE procedures according to the following scheme.

```

NEW(p)                = ALLOCATE(p,TSIZE(T))
DISPOSE(p)            = DEALLOCATE(p,TSIZE(T))
```

or, in case *p* points to a record with variant parts

```
NEW(p,t1,t2,...)      = ALLOCATE(p,TSIZE(T,t1,t2,...))
DISPOSE(p,t1,t2,...)  = DEALLOCATE(p,TSIZE(T,t1,t2,...))
```

The `TSIZE` procedure computes the number of storage units necessary to allocate an object of type *T*, possibly with the variant parts selected by *t1,t2,...* . The `TSIZE` evaluation is performed internally by the front end part of the Modula-2 compiler.

The result of an encounter with `NEW` or `DISPOSE` in a Modula-2 program will be that the intermediate code for a normal procedure call is generated by the front end after a check that these procedures are imported into the actual module, or explicitly programmed.

From this follows that neither the `NEW`, `DISPOSE` nor the `ALLOCATE`, `DEALLOCATE` procedure pairs are of any concern to the implementor.

4.3.2 System procedure specifications

The system procedure specifications will allow the implementation of low-level control sequences that are tailored to the hardware and the application.

```
$+  SystemProcDecl = SYSTEM PROCEDURE ident
$+                      [("(" TypeParamList ")") [":" Type]]
$+                      [BEGIN
$+                          ProcRuleSequence
$+                      END.

$+  ProcRuleSequence = (* see 4.3.1 *)
```

The type parameters may be of system or standard type. The procedure "ident" may be any not reserved identifier. Also standard procedure identifiers are permissible, as a parameter of system type will require that the implementation of that "standard" procedure is marked as a system object. In all other respects the specification of a system procedure is identical to a standard procedure specification.

A simple example of a system procedure specification is given by the SWAP procedure below.

Example

```
SYSTEM PROCEDURE SWAP(WORD): WORD;
BEGIN

    RULE <Expr> ::= SWAP ( <Expr> ) DO

        EmitRR(  ^XCHG^,    ^AL^, ^AH^ );

    END;
END;
```

In this example a code emitting procedure "EmitRR" is called to generate the symbolic assembler instruction in the parameter list. The "<Expr>" non-terminal indicates that the parameter has been evaluated and placed in the stack. In the implementation of the iAPX86 processor family the stack top is always contained in register "AX". The "XCHG" instruction in the example exchanges the contents of the byte halves "AH" and "AL" of that register. The result of a call to the SWAP procedure during run-time will thus be that the value of the actual parameter is placed at the stack top and there the contents of the byte halves is swapped and left for further use.

Another example is the following:

```
SYSTEM PROCEDURE OUT(port: CARDINAL, value: BYTE);
BEGIN

    RULE <SysProc> ::= OUT ( <Expr> , <Expr> ) DO

        Pop(^DX^);
        EmitRR(    ^OUT^,    ^DX^, ^AL^ );
        Remove;

    END;

    RULE <SysProc> ::= OUT ( <Const> , <Expr> ) DO

        EmitIR(    ^OUT^,    <Const>.value, ^AL^ );
        Remove;

    END;
END;
```

In the example above the two forms of the "OUT" instruction is utilized. Another code emitting procedure "EmitIR" is referenced

together with two stack handling procedures "Pop" and "Remove".

The examples demonstrate that the specification of standard and system procedures requires that code emitting and other support routines are available. Also, the designer of the specifications must have a basic understanding of the implementation model and the support procedures. In the next chapter the implementor module containing the complementary grammar rules, the attributes and the support procedures will be discussed.

Basic system procedures

The basic system procedures recommended in [Wirt82] to be included in any implementation are the following

ADR(x) Delivers the address of the variable x.

SIZE(v) Delivers the allocation size of variable v.

TSIZE(T) Delivers the allocation size of objects of type T.

Calls to the TSIZE procedure can be evaluated by the front end during compile time.

Calls to the SIZE procedure must be evaluated during run-time in case the parameter is of open array type, otherwise the size can be determined by the front end.

Calls to the ADR procedure must be evaluated during run-time since, apart from absolutely allocated objects, the only addresses known during compile time are offset addresses within allocation areas.

Implementation of the ADR and SIZE procedures

The ADR system procedure could be declared with an open array type of WORD elements as formal parameter type in order to achieve compatibility to all types.

First the ADDRESS type is declared:

SYSTEM TYPE LONGPOINTER = POINTER OF 32;

SYSTEM TYPE ADDRESS = LONGPOINTER TO BYTE;

The ADR can now be implemented according to the following declaration.

```

SYSTEM PROCEDURE ADR(VAR ARRAY OF WORD): ADDRESS;
BEGIN
    RULE <Expr> ::= ADR ( <Var> ) DO

        GetDS('BX',<Var>),
        Push('BX');
        EmitRM(  'LEA',    'BX',<Var>    );
        Push('BX'),

    END;
END;
```

The "GetDS" procedure will emit the instructions for loading the segment part of the "<Var>" address into the register "BX". The "LEA" instruction computes the offset part of the "<Var>" address. In the case of a structured parameter type the address to this operand must be computed during run-time. The "EmitRM" support procedure must be versatile enough to generate the proper code sequence. The ADR call will result in the pushing of the segment and offset parts of the operand address onto the stack.

The SIZE system procedure can be implemented as

```

SYSTEM PROCEDURE SIZE(VAR ARRAY OF WORD): CARDINAL;
BEGIN
    RULE <Expr>::=SIZE( <Var> ) DO

        Push('AX');
        IF <Var>.ArrayType THEN
            LowBound('BX',<Var>);
            HighBound('AX',<Var>);
            EmitRR(  'SUB',    'AX', 'BX' )
        ELSE
            EmitRI(  'MOV',    'AX',<Var>.size)
        END;
    END;
END;
```

The "LowBound" and "HighBound" procedures will generate the instructions necessary to obtain the bounds during run-time. The system procedure will also take care of operands with static size in case this is not already done by the front end.

The implementation of NEWPROCESS and TRANSFER will be discussed in chapter 9.

The implementation of IOTRANSFER will be discussed in chapter 10.

Implementation of type transfer functions

The type transfer functions are used to override the type compatibility control of the compiler. The type transfer may be applied to and from operands with a type that has the same allocation size. No conversion instructions will be emitted as in the case of type converting procedures such as FLOAT and TRUNC.

The compiler will allow standard type identifiers and imported system identifiers to be used as type transfer identifiers, i.e. user defined type identifiers will not be permissible. The compiler will check the allocation sizes and generate an error message if not equal.

4.4 SPACE ALLOCATION PREFIXES

Absolute allocation of variable objects within the address space is (in the PDP-11 implementation) accomplished by specifying the absolute address constant within square bracketed parentheses directly after the variable identifier in the declaration statement.

Example

```
VAR
    s[777560B]:    BITSET;
    x[777562B]:    CHAR;
```

The iAPX86 family provides a separate I/O address space in addition to the memory address space. Dedicated input and output instructions are used to transfer data to and from ports located in the I/O space.

The Z8000 family also has a separate I/O space with dedicated instructions in addition to the memory space. Furthermore, it has a third "special" I/O space primarily used to accommodate the memory management units.

The M68000 and NS16000 families have one address space only.

The Texas Instrument family has a somewhat unusual I/O logic. In addition to the memory space a separate I/O field of up to 4096 bits called the "Communication Register Unit" (CRU) is included. Within the CRU space single bits or sequences of bits can be addressed by special instructions.

The existence of several address spaces calls for a notation that will select the proper address space when a variable is allocated at a specific "absolute" address. To accomplish this it will be possible to define the presence and range of each address space as an object of subrange type.

```
$+ SpacePrefixSpec = SYSTEM SPACE ident "=" ident OF SubrangeType.
```

A space prefix is always declared as a system object implying that any importing module will be considered to be a implementation dependent low-level module.

Example

```
SYSTEM SPACE MEM      = MEM OF [0..OFFFFH];
SYSTEM SPACE IO       = IO  OF [0..17777BH];
SYSTEM SPACE CRU      = CRU  OF [0..4095];
SYSTEM SPACE SPIO     = SPIO OF [0..OFFFFH];
SYSTEM SPACE XMEM     = MEM  OF [10000H..OFFFFFH];
```

Provided it is imported, the address space identifier will be possible to use as a prefix to the absolute address constant when a variable object is located in a fixed location in one of the address spaces.

Example

```
FROM SYSTEM IMPORT MEM,IO,...;
. . . .
VAR
  MemLoc[MEM: 0776H]:  INTEGER;
  IoLoc[IO: 4]:        BYTE;
  CruLoc[CRU: 0]:      WORD;
  SpiLoc[SPIO: 0201H]: BYTE;
  XMemLoc[XMEM: 1777H]: BITSET;
```

The compiler will not reserve data space for variables allocated at fixed locations, neither will it check for double allocation. The only check performed is that the object is allocated within the limits of the subrange type.

CHAPTER 5. THE DESCRIPTION MODULES

As previously mentioned there are three kinds of description modules in the ACS; the implementor module, the component module and the configuration module. These modules can briefly be characterized as follows. The implementor module contains the implementation specific parts of the code generator for a certain target processor. The component modules contain the type, operator and procedure specifications associated to a certain component as described in the previous chapter. The configuration module specifies the set of component module objects to be included in the configuration information for the target configuration.

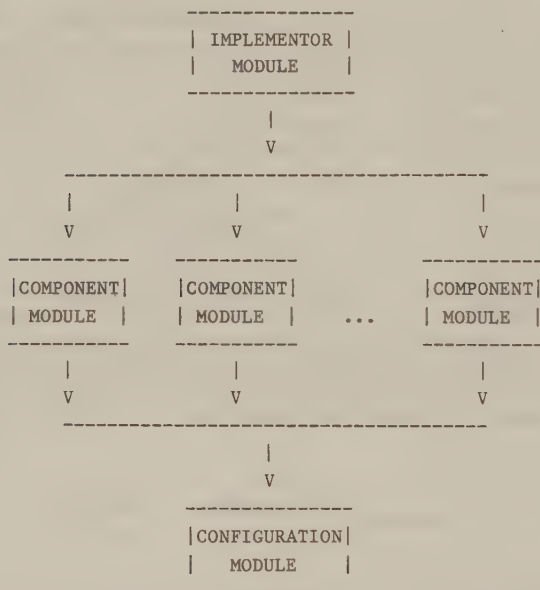


Fig 5.1

The description modules

In the following sections the description modules will be discussed in further detail.

5.1 THE IMPLEMENTOR MODULE

Compared with the component modules the implementor module represents a hidden level of the code generator implementation. While the component modules contain specifications of standard and system objects to be used at the programming level, the implementor module contains no such objects but rather specifications that are required to support the references and information in the component modules.

The structure of an implementor module corresponds to a Modula-2 module with the addition of terminal, attribute, code rule, and run-time procedure specifications.

The syntax is

```
$+ ImplementorModule = IMPLEMENTOR MODULE ident ";"
$+                               [export]
$+                               {import}
$+                               {declaration}
$+                               {TerminalSpec}
$+                               {AttributeSpec}
$+                               {ImplCodeRuleSequence}
$+                               {RunTimeProcedure}
$+                               [BEGIN
$+                               StatementSequence]
$+                               END ident "." .
```

The implementor module requires a processing pass to convert and to extract information concerning the non Modula-2 specifications.

The result of this pass is

- A main module containing pure Modula-2 code.
- A list of terminal symbols
- A collection of grammar rules with semantic specifications
- Extraction of the run-time procedures

The resulting Modula-2 program will constitute the main module of the code generator. It will be linked with the LALR(1) parser generated by the BOBS system when the code generator is formed. Optionally it may import objects from other Modula-2 compilation units.

5.1.1 Object declarations

The implementor module is supposed to define the stack handling and code emitting procedures and the attribute objects referenced by the code template statements in the component modules. In addition to these objects the implementor module may contain declarations of other Modula-2 objects to be used within the implementor module only. All object references obey the standard Modula-2 scope rules. The statement part of the code rules may reference objects in the same way as the statement part of a procedure with the addition of attribute references as described in section 4.2. Local modules may be used within an implementor module and are recommended for isolating the definition of different categories of support objects.

5.1.2 Attribute declarations

Attributes are variable elements associated with the non-terminals of the grammar. When a certain reduction is performed by the parser the attributes of the non-terminals involved can be referenced in the statement part of the actual code rule. The BOBS generated parser will maintain a stack where the topmost stack elements correspond to the right hand side symbols in the production, the terminal symbols included. The non-terminal to the left shares the attribute record with the leftmost symbol on the right hand side. Before a reduction is applied the attribute elements can be used as arguments, in expression evaluation or in assignments. In particular the attributes of the leftside non-terminal can be assigned values to be transferred up the parsing tree, thus being made available for reference in the subsequent reductions. Such attributes are called "synthesized" in contrary to "inherited" attributes transferred in the opposite direction, i.e. down the tree. Inherited attributes are not supported by the BOBS parser and hence not by the compiler system.

Two kinds of attribute declarations are possible: common and private attributes. The common attributes will be available for reference by all non-terminals, while the private attributes should be referenced by their associated non-terminal only.

The syntax for the attribute declarations is

```
$+ AttributeDecl = CommonAttributeDecl | PrivateAttributeDecl.  
$+ CommonAttributeDecl = COMMON ATTRIBUTE ident "=" SimpleType.  
$+ PrivateAttributeDecl = PRIVATE ATTRIBUTE NonTerminal "=" RecordType.
```

All the attribute identifiers must be unique. This is true also for the attribute identifiers within the private record types. The reason for this is that the declared attributes will be transformed by the implementor module processor to a single record type with the common attributes in a common part and the private attributes in variant parts corresponding to the declared attribute records.

Example

```
COMMON ATTRIBUTE value:  ARRAY [0..3] OF INTEGER;

PRIVATE ATTRIBUTE <Var> =  RECORD
    AllocKind:      (DA,LR,LVP,LCP,...);
    level:          [0..maxlevel];
    reladdr:        CARDINAL;
    size:           [0..maxsize];
    END;
```

5.1.3 Terminal symbols

A terminal symbol within the grammar production part of a rule must either be a special character of Modula-2 or it must be explicitly defined in a terminal specification. Specified terminal symbols must be identifiers only.

The syntax for a terminal declaration is as follows.

```
$+ TerminalSpec = TERMINAL Terminallist.
$+ Terminallist = Terminal {", " Terminal}.
```

Example

```
TERMINAL
    ENTERPROG,BEGINPROG,EXITPROG,
    MODULE,BEGINMOD,ENDMODULE,
    PROCEDURE,BEGINPROC,ENDPROCEDURE,
    EXIT,RETURN,%DO,%END,...
```

If the "DO" and "END" symbols are included they must be preceded by a "%" not to be confused by the "DO" and "END" symbols of the rule.

5.1.4 Grammar and code rule specifications

The grammar and code rules required for the full implementation of the code generator should be specified in the implementor module. The syntax is similar to the code rules for the operator and procedure specifications in the component modules. The difference is that the format of the grammar production is allowed to be more general.

```
$+  ImplCodeRuleSequence = ImplCodeRule {";" ImplCodeRule}.
$+  ImplCodeRule = RULE ImplProduction [DO
$+                StatementSequence]
$+                END.
$+  Production = NonTerminal "::~=" SymbolList.
$+  SymbolList = Symbol {Symbol}.
$+  Symbol = NonTerminal | Terminal | EMPTY.
```

The code rules in the implementor module will in many cases contain the production part only.

Example

```
RULE <CompUnit> ::= <EnterProg> <Block> <ExitProg> END;
```

The statement part of the code rules in the implementor module may contain references to any Modula-2 object accessible within the implementor module and also references to declared attribute elements.

Together with the productions in the component modules the productions specified in the implementor module must define a complete unambiguous LALR(1) grammar. The BOBS parser will check this.

5.1.5 Run-time procedures

Modula-2 is designed with the aim to require minimal support by a run-time system. Nevertheless, there are certain code sequences that are reasonable to include in a central run-time system.

For example, often used standard or system procedures that need more than a few lines of code for their computation should be possible to include in the run-time system for space saving reasons. This is valid also for the error checking routines.

Another example is the initiating routine of a Modula-2 execution. There is a need for such a routine to define the initial processor

state and to set up the main process and start the execution of the initiating statement parts of the linked-in implementation modules. It is possible to generate this sequence by a built-in procedure in the code generator, but it seems more practical if this kind of routine could be defined in assembler notation together with the data referenced, and then included in the run-time system.

The assembler code will be collected as it stands with no alterations. However, in order to make a safe collection the assembly code should be surrounded by comment parentheses.

```
$+ RunTimeProcedure = RUNTIME PROCEDURE ident ";"  
$+ BEGIN  
$+      "(*" AssemblerStatements "*")"  
$+ END ";".
```

The implementor module processor will collect the assembler code, submit it for translation by the standard assembler, and then transform the result to a form that is linkable with the translated Modula-2 modules.

Example

```
RUNTIME PROCEDURE Check;  
BEGIN  
  (*          CSEG  
Check:  
    ; Range checking routine      ;  
    ; AX: check value             ;  
    ; BX: low limit               ;  
    ; DX: high limit              ;  
  
    CMP    AX,BX      ; (AX) >= (BX)?  
    JN     CheckErr  
    CMP    DX,AX      ; (DX) >= (AX)?  
    JN     CheckErr  
    RET                     ; Return  
    ; Issue an Overflow interrupt  
CheckErr:  
    MOV    AX,1        ; Error indicator  
    INT    3           ; Type 3 interrupt  
    ; < Run Overflow process>  
    RET                     ; Return after interrupt  
  *)  
END;
```

The example demonstrates the specification of a range checking routine to be included in the run-time system. In case the check fails the routine will issue a type 3 (overflow) interrupt that will transfer the execution to an interrupt process presumably programmed at the Modula-2 level, otherwise the calling routine will be immediately resumed.

The call to the range checking routine is generated by a code generating procedure invoked within a code rule.

Example

```
RULE <Index> ::= INDEX <ExprS> <lowlimit> <highlimit> <ixsize> DO
```

```
    EmitRI(  "MOV",    "BX",<lowlimit>.value  );
    EmitRI(  "MOV",    "DX",<highlimit>.value  );
    CallRTS("Check");
    EmitRR(  "MOV",    "DX",<ixsize>.value      );
    EmitRR(  "IMUL",   "AX","DX"                );
```

```
END;
```

The "CallRTS" procedure in the example above must be coded to generate a run-time procedure call to the referenced entry. The calling sequence naturally depends on the run-time organization of an executing Modula-2 program. In most cases the call will be indirect via a link table requiring that a binding between the entry number and the entry name must be established at a later point.

5.2 THE COMPONENT MODULE

The component module contains the operator, procedure and space prefix specifications as described in the previous chapter. Each component with processing capabilities should be described in a separate component module. Also components with non-processing capabilities but requiring special code sequences when they are used should be described in a corresponding component module.

Apart from a syntactic check of the specifications, the task of the component module processor is to generate component information to be used by the configuration module processor when the configuration information for a new Modula-2 compiler is produced.

The following tasks will be performed by the component module processor when an operator/procedure specification is processed:

- a) in case of an operator specification an internal operator identifier will be generated,
- b) the type and operand information in the operator/procedure specification head will be collected and written to a resulting information file,
- c) the grammar rules will be modified according to the standard expected by the code generator and the operator/procedure identifier will be inserted in a postfix position,
- d) the modified grammar rules will be written to a resulting grammar file,
- e) the non-terminal references within the code templates will be replaced by proper references according to the attribute record in the implementor module,
- g) the code template statements will be written to a code file to be included in the code generator.

The information file

For each operator specification the following information will be registered in the information file:

-	Internal operator identifier	Ex: addxx
-	Associated Modula-2 operator	Ex: +
-	Resulting operand type (if any)	Ex: SHORTINTEGER
	Basic type	Ex: INTEGER
	Allocation size	Ex: 8
-	Number of operands	Ex: 2
-	For each operand:	
	Type	Ex: SHORTINTEGER
	Basic type	Ex: INTEGER
	Allocation size	Ex: 8

The exemplifying values to the right refers to the following operator specification.

```

SYSTEM OPERATOR (SHORTINTEGER + SHORTINTEGER): SHORTINTEGER,
BEGIN
    ...
END;

```

The procedure information resulting from a standard/system procedure specification in a component module corresponds to the operator information. The specified procedure identifier will be used also internally.

-	Internal procedure identifier	Ex: ABS
-	Associated Modula-2 standard procedure	Ex: ABS
-	Resulting operand type (if any)	Ex: INTEGER
	Basic type	Ex: INTEGER
	Allocation size	Ex: 16
-	Number of operands	Ex: 1
-	For each operand:	
	Type	Ex: INTEGER
	Basic type	Ex: INTEGER
	Allocation size	Ex: 16

The exemplifying values refers to the procedure specification:

```

STANDARD PROCEDURE ABS (I: INTEGER): INTEGER;
BEGIN
    ...
END;

```

The front end will read the information file each time the Modula-2 compiler is invoked and organize the information internally. When a Modula-2 operator/procedure is encountered in the source code the front end will consult the information to find a match to the operand types. If a match is found the corresponding internal identifier will be used in the intermediate representation.

The grammar file

The component module processor will collect the grammar rules from the operator/procedure specifications into a resulting grammar file. The previous operator specification is expanded to demonstrate the collection and modification of the grammar rules.

```

SYSTEM OPERATOR (SHORTINTEGER + SHORTINTEGER): SHORTINTEGER;
BEGIN
    RULE <Expr> ::= <ExprS> + <Expr> DO
        / Code Template 1 /
    END;

    RULE <Expr> ::= <Expr> + <Constant> DO
        / Code Template 2 /
    END;

    RULE <Expr> ::= <Expr> + <Var> DO
        / Code Template 3 /
    END;
END;

```

The operator specification above will result in three productions for the grammar file using the notation required by the BOBS system. The "+" operator will be replaced by a postfix internal operator used in all the three resulting grammar rules. In addition a unique production number is generated for each of the grammar rules.

```

"Expr" | n1 | "ExprS" "Expr" addxx
"Expr" | n2 | "Expr" "Constant" addxx
"Expr" | n3 | "Expr" "Var" addxx

```

As a consequence of the information file read by the front end the internal operator "addxx" will be used by the front end whenever intermediate code for addition between operands of "SHORTINTEGER" type is to be generated and thus a binding between an operand pair in the intermediate representation and a set of possible grammar rules is established.

The code file

The component module processor will prepare a code file with the code templates labeled with the production numbers corresponding to the associated grammar productions.

According to the function of the BOBS parser the production number of a grammar rule will be referenced as a parameter when a semantic routine is called by the parser in the code generator when that grammar rule is applied for a reduction. If the code templates are

included in the semantic routine as alternatives in a case statement and labeled with the production number of the associated grammar rule they will be executed when that reduction takes place.

```
CASE N OF
```

```
...
```

```
n1: / Code Template 1 /
```

```
n2: / Code Template 2 /
```

```
n3: / Code Template 3 /
```

```
...
```

```
END;
```

The component module processor will not check the semantics of the code template part of a rule, except for the non-terminal references. Instead this will be performed when the resulting code generator part of the Modula-2 compiler is compiled.

It is the task of the configuration module processor to make the inclusion of code templates belonging to component objects (operators and procedures) specified in the configuration module. Objects declared within a component module must appear in the export list of the component module in order to be available for reference by the configuration module that will be discussed next.

5.3 THE CONFIGURATION MODULE

The configuration module defines the set of component module objects that should be included in the component information used when the corresponding Modula-2 compiler is generated. The reason for a separate module for this purpose is to provide a flexible and visually clear interface to the user of the ACS.

The syntactic specification for the configuration module is

```
$+ ConfigurationModule = CONFIGURATION MODULE ident ";"
$+                       {import}
$+                       END ident "." .
```

The import lists are specified in standard Modula-2 fashion. Either all objects exported from a component module are imported by

specifying the name of the component module in the import list, or objects are imported explicitly from a certain component module.

Example

```
CONFIGURATION MODULE Modula286;

IMPORT iAPX286;

FROM NDP80287 IMPORT
    REAL, LONGREAL, SHORTINTEGER, LONGINTEGER,
    FLOAT, TRUNC,
    PI, Sqrt, ...

FROM OS80130 IMPORT
    CreateJob,
    CreateTask, DeleteTask, SuspendTask, ...,
    CreateMailBox, DeleteMailBox, ReceiveMessage, ...

END Modula286.
```

When a type object is imported all operations defined for that type are also imported. Operations defined for mixed operand types will be imported only if both types are imported. Objects of standard type imported into the configuration module will become predeclared in all modules compiled by the generated Modula-2 compiler. Objects of system type will be included in the pseudo module SYSTEM, i.e. such objects must be explicitly imported into the Modula-2 modules referencing them.

The processing of the configuration module will result in the creation of information corresponding to the specified configuration, i.e. the configuration files containing procedure, operator and operand type information for the front end. The grammar rules and the code rule statements for the generation of the back end will be collected from the component information files and complemented by the grammar rules and routines from the implementor module before the generation of the corresponding code generator takes place.

The description modules must be processed in the order of import/export dependence, i.e. first the implementor module, then the component modules and last the configuration module(s).

CHAPTER 6. THE INTERMEDIATE REPRESENTATION

The intention behind the specifications discussed in the previous chapters was to obtain control of the code sequences emitted by the code generator for the specified operations and procedure calls. The first step to achieve this was to insert internal operators in the intermediate representation (IR) and into the corresponding grammar productions to be used by the code generating phase. A substantial part of the grammar for the intermediate representation will consist of productions derived from such specifications. The remaining productions concern the operand addressing, control statements, normal procedure calls, procedure and program entries and exits, procedure parameter pushing, run-time checks, and the reduction of the operands to the non-terminals referenced in the production parts of the code rule specifications.

The IR is a linearized sequence of operators, operands and associated code generation information in postfix form. Type checking and storage binding has been performed but not register allocation nor any optimization. The design of the fix part of the IR is not crucial for the function of the adaptable compiler system but rather for the possibility to generate efficient code.

The parts of the IR presented in this chapter concern the addressing of operands and the representation of structured variables and procedure calls.

6.1 ADDRESSING

6.1.1 Address modes

The address mode describes the way an operand is accessed. An operand can be located in a global data area, in a constant area, within a procedure frame in the work space of a process, or it can be an immediate constant. In addition to this it may have the more complicated access path of an indexed variable requiring a number of index computations during run-time. A global operand may also be external to the module from which it is referenced.

The address mode of each operand is described in the intermediate representation (IR) produced by the front end of the Modula-2 compiler. In the code generating part of the compiler the corresponding address mode productions must be specified as well as the rules for the address computations in the target processor.

6.1.2 Allocation areas

The front end of the Modula-2 compiler makes certain built in assumptions concerning the logical organization of a Modula-2 program. In the storage allocation process it will assume separate allocation areas for the following objects in each separately compiled module.

- | | | | |
|----|------------------------------|----|------------------|
| a) | Global variables | => | "Data Area" |
| b) | Constants | => | "Constant Area" |
| c) | Procedures and module blocks | => | "Code Area" |
| d) | Local variables | => | "Local Area" |
| e) | Local parameters | => | "Parameter Area" |

The "Data Area" will accomodate the variable objects at static nesting level 0 in each separately compiled module. The "Constant Area" is primarily intended to contain string literals and other constant objects that are not suitable for representation by immediate values in the code. The "Code Area" will contain the code bodies of procedures and module initiating blocks. For the local objects a new "Local Area" and a new "Parameter Area" is associated to each procedure. The "Parameter Area" is intended for the actual/formal parameter interface and the "Local Area" will contain variables local to the procedure.

6.1.3 Prefixes

In the intermediate code the allocation of operands in the various areas is indicated by prefixes followed by supplementary addressing information.

DA reladdr

Data Area Reference.

The operand is allocated in the "Data Area" at offset "reladdr".

LA reladdr

Local Area Reference.

The operand is allocated at offset "reladdr" in the "Local Area" of the current procedure frame.

LVP reladdr	Local Variable Parameter Reference. The operand is referenced by offset "reladdr" in the "Parameter Area" of the current procedure frame.
LCP reladdr	Local Constant Parameter Reference. The operand is referenced by offset "reladdr" in the "Parameter Area" of the current procedure frame.
IMR level reladdr	Intermediate Reference. The operand is referenced by offset "reladdr" in the "Local Area" of a procedure frame at the intermediate static nesting level "level".
IMVP level reladdr	Intermediate Variable Parameter Reference. The operand is referenced by offset "reladdr" in the "Parameter Area" of a procedure frame at the intermediate static nesting level "level".
IMCP level reladdr	Intermediate Constant Parameter Reference. The operand is referenced by offset "reladdr" in the "Parameter Area" of a procedure frame at the intermediate static nesting level "level".
CON reladdr	Constant Reference. The operand is referenced by offset "reladdr" in the "Constant Area".

6.1.4 Access Paths

The access paths to the operands are defined in the fix IR productions. An operand reference can be categorized as either a "Constant", a "Simple Variable", an "Array Element", a "Record Element" or a "Direct Reference".

```

<Var> ::= <Constant> | <SimpleVar> | <ArrayElem>
        | <RecordElem> | <DirectRef>

```

Constants

There are two kinds of constants represented in the IR; constants allocated in the "Constant Area" and immediate constants. The "Constant Area" is supposed to contain constants not suitable for reference as immediate operands in the target instructions, such as strings and floating point literals.

```
<Constant> ::= CON <reladdr> <size>
              | IMM <immvalue> <immtype>
```

The "<immvalue>" is a sequence of literal values. The concluding "<immtype>" indicates if the literals should be interpreted as a integer or cardinal value.

Simple Variables

The definition of a simple variable starts with the following productions.

```
<SimpleVar> ::= VAR <BasicRef> <size>
```

```
<BasicRef> ::= <LogRef> | <AbsRef>
```

The "<BasicRef>" specifies either a logical reference or an absolute reference. Absolute references will occur as a result of references to absolutely allocated variables, i.e. variables declared with a space prefix and an absolute address specified by the programmer. The "<size>" holds the size of the variable in bytes.

Logical references

References to (global) variables may be indicated as external, i.e. imported from another module. The logical access path will in this case start with a LINK prefix followed by a unit number. The unit number indicates the module from which the variable object is imported.

```
<LogRef> ::= <Link> <BitRef> <LogAddr>
```

```
<Link> ::= LINK <unit> | EMPTY
```

```
<BitRef> ::= BIT <lowbit> <highbit> | EMPTY
```

The "<LogAddr>" part of the reference will contain information regarding the allocation area and relative address to the variable within this area.

```

<LogAddr> ::= DA <reladdr> |
              LR <reladdr> |
              LVP <reladdr> |
              LCP <reladdr> |
              IMR <level> <reladdr> |
              IMVP <level> <reladdr> |
              IMCP <level> <reladdr>

```

Absolute references

The intermediate representation of absolute references will include the space prefix followed by the absolute address.

```

<AbsRef> ::= ABS <Prefix> <absaddr>

```

The front end will check that the absolute address falls within the range specified in the prefixed space declaration.

Address information attributes

The code generating part of the compiler will collect the addressing information during the parsing of the IR and successively store it in the attributes associated to the non-terminals.

```

Link:          BOOLEAN;
unit:          [0..maxunit];
BitRef:        BOOLEAN;
lowbit:        [0..maxbit];
highbit:       [0..maxbit];
AllocKind:     (DA,LR,LVP,LCP,IMR,IMVP,IMCP,CON,IMM,ABS);
level:         [0..maxlevel];
reladdr:       CARDINAL;
size:          [0..maxsize];
length:        [0..maxlength];
prefix:        prefixtype;
absaddr:       CARDINAL;

```

Array Elements

An array element is represented according to the following productions.

```
<ArrayVar> ::= ARRAY <BasicRef> [ <IndexList> ] <elemsize>
<IndexList> ::= <Index> , <Index> | <Index>
```

Access to the elements of an array requires that the index values are computed during run-time. An exception is the index lists containing only constant index values, since in this case the index evaluation can be performed entirely during compile time.

The IR representation of array indexes is as follows.

```
<Index> ::= INDEX <Expr> <lowlimit> <highlimit> <ixsize> |
          <ConstIndex>
```

The <lowlimit> and <highlimit> symbols are intended for checking that the index value is within the permitted range. The <ixsize> symbol holds the byte length of the array element, i.e. the value to be multiplied to the index value in order to obtain the valid offset.

Example

```
RULE <Index> ::= INDEX <Expr> <lowlimit> <highlimit> <ixsize> DO
```

```
    EmitRI(  'MOV',    'BX',<lowlimit>.value    );
    EmitRI(  'MOV',    'DX',<highlimit>.value    ),
    CallRTS('Check');
    EmitRR(  'MOV',    'DX',<ixsize>.value);
    EmitRR(  'IMUL',   'AX','DX');
```

```
END;
```

In the case of more than one index the results are successively accumulated when the index list rule is applied. Sequences of constant indices should be evaluated during compile time.

Examples of array information attributes:	ArrayType:	BOOLEAN;
	lowlimit:	INTEGER;
	highlimit:	INTEGER;
	offset:	ADDRESS;
	elemsize:	CARDINAL;

Record Elements

The offset to a field in a record element is prefixed by a OFFSET indicator in the IR.

```
<SimpleVar> ::= <SimpleVar> OFFSET <offset>
<ArrayVar> ::= <ArrayVar> OFFSET <offset> |
               <SimpleVar> OFFSET <offset> "[" <IndexList> "]" |
               <ArrayVar> OFFSET <offset> "[" <IndexList> "]"
```

The rules above corresponds to the a.b, A[i,...].b, a.B[j,...] and A[i,...].B[j,...] forms respectively.

Block References

Modula-2 allows direct assignment between array or record elements on condition they are of the same type. Elements referenced on a block basis are in the IR indicated by a BLOCK symbol followed by the size of the block in bytes. Block references may also occur in an actual parameter list. In most processors block movement can be implemented by special instructions designed for this purpose.

```
<Assignment> ::= <SimpleVar> BLOCK <size>
                  <SimpleVar> BLOCK <size> :=
```

6.3 PROCEDURE CALLS

The intermediate representation of normal procedure calls is defined by the following productions.

```
<ProcCall> ::= PROCCALL "(" <ParamList> ")" <MemRef> |
               PROCCALL "(" <ParamList> ")" <EntryRef>
```

```
<ParamList> ::= <ParamList> "," <Param> | <Param>
<Param> ::= VAR <Var> | VAL <Expr>
```

The code to push the actual parameters onto the stack will be generated while the reduction to the "<ParamList>" non-terminal takes place. Depending on the reference type an address or an evaluated parameter will be pushed.

The grammar productions for the standard and system procedure calls are included in the grammar for the IR by the configuration module

processor. Each specified standard and system procedure has its own production rule with a unique identifier associated to the corresponding code template.

```
<StdProc> ::= STDCALL "(" <StdParamList> ")" <ident>  
<SysProc> ::= SYSCALL "(" <SysParamList> ")" <ident>
```

```
<StdParamList> ::= <StdParamList> "," <StdParam> | <StdParam>  
<StdParam> ::= STDVAR <Var> | STDVAL <Expr>
```

```
<SysParamList> ::= <SysParamList> "," <SysParam> | <SysParam>  
<SysParam> ::= SYSVAR <Var> | SYSVAL <Expr>
```

The processing of the standard and system parameters depends on the associated code templates in the implementor module. Compared to the processing of a normal parameter list there is no need for a standardized interface to compiler generated code. On the contrary it is an important aspect of the system procedures that the parameter interface is allowed to be tailored to the intended use of the emitted code.

CHAPTER 7. RUN-TIME ORGANIZATION

7.1 DESIGN CONSIDERATIONS

The run-time organization of a Modula-2 program should reflect the design philosophy of the language, i.e. the required run-time routines should be few and efficiently implemented, large programs with many separately compiled modules should be supported.

7.1.1 Run-time routines

The run-time routines required by any implementation are the following:

- a) An initiating routine to set the initial register values and then controlling the execution of the initiating statement blocks in the implementation modules before the execution of the main program statements starts.
- b) The central parts of the standard and system procedures not implemented in straight code.
- c) Error handling routines invoked by range and exception checks in the generated code sequences.
- d) A termination routine to be invoked at the end of the main program, or when a process executes the END statement of its start-up procedure.

Depending on the implementation additional run-time routines may be required. For example routines to support debugging, or execution analysis.

In case the processor supports range and exception checks on an interrupt basis the error handling may be programmed entirely in Modula-2.

The implementation of some of the standard and system procedures may cause a conflict between execution speed and code minimizing. The compiler system will allow the generation of a run-time system adapted to the actual application also in this respect.

7.1.2 Linked references

References between separately compiled modules can be solved in basically two ways. Either each linked reference is modified at link time, or linked references are solved indirectly during run-time. The former approach results in faster execution and saves the space required for the indirect link-tables. On the other hand the indirect approach is attractive in other respects. First, it allows reorganization and other manipulations of the modules during run-time. Second, the indirect reference approach is supported by the new processor generation (NPD16000,iAPX286) where the link tables make it possible to write very large programs and to utilize the protection and virtual management facilities.

In the following only the indirect reference approach will be considered.

7.1.3 Storage areas

The areas to be allocated for each compiled module are:

The "Data Area" containing global variables.

The "Constant Area" containing strings and constant objects not suitable for representation by immediate values.

The "Code Area" containing the procedure and module statement blocks.

The "Table Area" containing link information.

For each program the following system areas must also be allocated:

A "Data Area" for the run-time routine data.

A "Constant Area" for the run-time routine constants.

A "Code Area" for the run-time routine code.

A "Table Area" for the central link tables.

A "Work Space" for the main process.

The following sections will demonstrate the implementation of these areas in the iAPX86 family.

7.2 RUN-TIME ORGANIZATION IN THE iAPX86

The iAPX86 architecture includes four segment registers

DS	-	Data Segment register
CS	-	Code Segment register
ES	-	Extra Segment register
SS	-	Stack Segment register

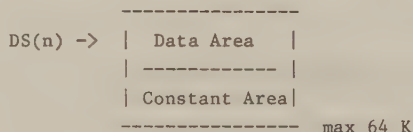
The locations within a segment is referenced by an offset address in the range 0-FFFFH, i.e. a segment contains at most 64 K bytes. A full address in the iAPX86 consists of the segment pointer left shifted 4 bits and added to the offset resulting in a 20 bit address able to reference up to 1 M byte of memory. The four bit shift of the segment pointers implies that the storage areas must be allocated at 16 byte boundaries.

7.2.1 Local storage disposition

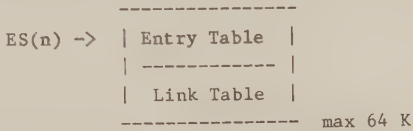
For each separately compiled module the "Code Area" is referenced by the CS register.



The "Data Area" and "Constant Area" are concatenated and referenced by the DS register.



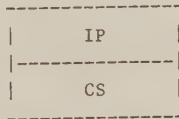
The "Table Area" will contain two tables, the "Entry Table" and the local "Link Table" referenced by the ES register.



The "Entry Table" contains an entry for each procedure and module block in the compilation unit. The first entry defines the entry point for the module initiation sequence, i.e. the first module statement block at top level.

Each node in the entry table contains an Instruction Pointer (IP) and a Code Segment Pointer (CS). The Code Segment Pointer has the same value, i.e. the "Code Area" segment address, in all entry nodes.

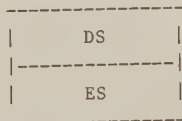
Entry Table node



The local "Link Table" contains an entry for each module referenced in an import statement. The first entry always refers to the run-time system part. The order of the other nodes is determined by the first occurrence of a reference to a module in an import statement.

Each node in the link table contains the Data Segment pointer (DS) and the Extra Segment pointer (ES) to the "Data Area" and "Table Area" in the external module.

Local Link Table node



A reference to an external procedure will require the following steps:

```
PUSH    ES                ; Save ES
PUSH    DS                ; Save DS
```

< Push parameters >

```
MOV      DS,ES: LocalLinkTable+4*unit    ; Load external DS
MOV      ES,ES: LocalLinkTable+4*unit+2  ; Load external ES
```

```
CALLF    ES: 4*entry        ; Call via external Entry Table
```

< Pop return value - if any >

```
POP      DS                ; Restore DS
POP      ES                ; Restore ES
```

A reference to external data will look like:

```
PUSH     DS                ; Save DS

MOV      DS,ES: LocalLinkTable+4*unit    ; Load external DS

MOV      AX,DS: offset      ; Load data word

POP      DS                ; Restore DS
```

7.2.2 System storage disposition

As stated previously the system storage should allocate space for the "Data", "Constant" and "Code" areas for the various run-time routines. In addition it should accommodate the central link table and also the workspace for the main process.

The run-time routines should be defined in the implementor module as described in section 5.1 .

For each linked program a central "Link Table" is created by the linker. It contains one entry for each linked compilation unit. The first entry refers to the run-time system and the order of the other entries is determined by their linking order. The central link table is used when a program is initiated.

Each node in the central link table contains the Data Segment pointer (DS) and the Extra Segment pointer (ES) to the "Data Area" and "Table Area" of the corresponding compilation unit.

Central Link Table node

	DS	

	ES	

CHAPTER 8. SPECIFICATION OF A NUMERIC COPROCESSOR

In this chapter the component module specifications for the Intel 8087 Numeric Data Processor will be discussed. The purpose is to demonstrate how the type, operator and procedure specifications are defined for a real component, and how limitations and irregularities in the component design can be handled by operation and procedure specifications.

8.1 TYPE SPECIFICATIONS

The Numeric Data Processor, in the following referred to as the NDP, implements the following data types.

SYSTEM TYPE WORDINTEGER	=	INTEGER OF 16;
SYSTEM TYPE SHORTINTEGER	=	INTEGER OF 32;
SYSTEM TYPE LONGINTEGER	=	INTEGER OF 64;
STANDARD TYPE REAL	=	REAL OF 32;
SYSTEM TYPE LONGREAL	=	REAL OF 64;
SYSTEM TYPE TEMPREAL	=	REAL OF 80;

In addition it also implements a packed BCD data type that will not be discussed.

The 32 bit real data type has been chosen to be the REAL standard type. The other data types are specified as system types implying that each of them must be imported from the SYSTEM module before reference.

8.2 OPERATOR SPECIFICATIONS

For each of the declared standard and system types the associated set of operators is expected to be declared. The NDP is organized as a stack processor with 8 internal 80 bit registers realizing the stack. In our implementation model the NDP will operate as a classical stack machine. Either the two topmost registers will be implicitly referenced by the operation, or one operand in memory will be explicitly specified together with an implicit reference to the top register. In either case the result will be left in the top of the

NDP register stack. The internal format of operands in the register stack is always 80 bit temporary real. Operands of other types are converted to this format when loaded to the stack, and converted back when stored in memory.

The specification of NDP operators is similar to the iAPX86 operators discussed previously. However, the fact that the NDP does not use the workspace stack of the current process, but rather an internal hardware stack, requires the definition of a non-terminal that is associated with the NDP stack.

Example

```

STANDARD OPERATOR (REAL + REAL): REAL;

BEGIN
    RULE <NDP> ::= <NDP2> + <NDP> DO

        Emit(      'WAIT'      );
        Emit(      'FADD'      );
    END;

    RULE <NDP> ::= <NDP> + <Var> DO

        Emit(      'WAIT'      );
        EmitNDPM( 'FADDP',Real32,    <Var>    );
    END;

    RULE <NDP> ::= <NDP> + <Constant> DO

        Emit(      'WAIT'      );
        EmitNDPC( 'FADDP',Real32,    <Constant>);
    END;
END;

```

In the example above the "<NDP>" and "<NDP2>" non-terminals represents the top resp. second element in the NDP register stack. In the first code rule a simple addition of the two top operands is performed leaving the result in the top register.

In the second code rule a source operand is fetched directly from memory and added to the stack operand. In this case the type, i.e. "Real32", of the memory operand must be specified as the required format conversion must be encoded in the generated NDP instruction. The third code rule is similar to the second. In this case a constant is specified as a source operand. The emitted instruction is the same

as in the previous case since a 64 bit constant will be allocated in the "Constant Area" in memory. The NDP is not capable of loading immediate values other than the built-in special mathematical constants. The preceding "WAIT" will be explained in the next section.

```
STANDARD OPERATOR (REAL := REAL): REAL;
```

```
BEGIN
```

```
    RULE <Var> ::= <NDP> DO
```

```
        Emit(      "WAIT"                                );
```

```
        EmitNDPM(  "FSTP",Real32,      <Var>              );
```

```
    END;
```

```
END;
```

The component module code rules in the examples must be supported by the following reductions in the implementor module that force the leftmost "<Var>" and "<Constant>" in a binary operation onto the NDP register stack.

```
    RULE <NDP> ::= <Var> DO
```

```
        Emit(      "WAIT"                                );
```

```
        EmitNDPM(  "FLD ",Real32,      <Var>              );
```

```
    END;
```

```
    RULE <NDP> ::= <Constant> DO
```

```
        Emit(      "WAIT"                                );
```

```
        EmitNDPM(  "FLD ",Real32,      <Constant>        );
```

```
    END,
```

Also a code rule that reflects the transfer of the top stack operand to the second top stack element must be defined. No instructions are emitted by this rule.

```
    RULE <NDP2> ::= <NDP> DO
```

```
    END;
```

The supporting grammar rules and code emitting procedures must be included in the implementor module.

8.2.1 Synchronization

Although the intention is that the NDP should be able to operate concurrently with the main processor (CPU) this objective is limited in two ways.

- (1) The CPU must not attempt to access a memory operand that has been read or written by a previous NDP instruction until that instruction has completed.
- (2) An NDP instruction that is processed by the NDP execution unit, i.e. all arithmetic, comparison, transcendental, constant and data transfer instructions, must not be started by the CPU if the NDP is still busy executing the previous execution.

The first synchronizing problem can be solved either by inserting a "WAIT" instruction after each memory referencing NDP instruction, or by inserting a "WAIT" before any CPU instruction accessing an NDP operand. The second synchronizing problem enforces the insertion of a "WAIT" instruction before any NDP instruction, except the (8) instructions not utilizing the NDP execution unit.

The "WAIT" instruction will cause the CPU to wait until the NDP is not executing. It is encoded in one byte and the execution time is $3+5n$ clock cycles, where n is the number of times the CPU have to examine if the NDP is busy.

The synchronization restrictions severely reduce the concurrency of the processors. Especially if the "WAIT" instruction is inserted after an NDP instruction. Therefore the synchronization problem should be solved by insertion of the "WAIT" instruction before any CPU instructions referencing operands with a type defined by the NDP component module.

The strong typing rules of Modula-2 guarantee that NDP and CPU operands can not be freely intermixed. The two cases where the CPU may be involved in NDP operand references are:

- (a) if a type transfer function is applied to an operand,
- (b) if an NDP operand is used as a parameter in a procedure call.

The definition of type transfer functions for NDP operands will be discussed next.

8.3 PROCEDURE SPECIFICATIONS

Any Modula-2 compiler supporting the REAL standard type will require the implementation of the FLOAT and TRUNC type conversion routines. These procedures can be implemented in straight code by using the built in type conversion of the NDP.

Example

```
STANDARD PROCEDURE FLOAT(x: INTEGER): REAL;
```

```
BEGIN
```

```
  RULE <Expr> ::= FLOAT ( <Expr> ) DO
```

```
    Push(`AX`);
    Push(`AX`);
    EmitRR(  `MOV`,          `BX`, `SP` );
    Emit(    `WAIT`                );
    EmitNDPA( `FILD`, Int16,  `SS: [BX]` );
    Emit(    `WAIT`                );
    EmitNDPA( `FSTP`, Real32, `SS: [BX]` );
    Pop(`AX`);
```

```
  END;
```

```
END;
```

```
STANDARD PROCEDURE TRUNC(x: REAL): INTEGER;
```

```
BEGIN
```

```
  RULE <Expr> ::= TRUNC ( <Expr> ) DO
```

```
    Push(`AX`);
    EmitRR(  `MOV`,          `BX`, `SP` ),
    Emit(    `WAIT`                );
    EmitNDPA( `FLD`, Real32,  `SS: [BX]` );
    Emit(    `WAIT`                );
    EmitNDPA( `FISTP`, Int16, `SS: [BX]` );
    Pop(`AX`);
```

```
  END;
```

```
END;
```

The NDP can load and store operands in memory only. The initial "Push" operation(s) will transfer the top stack element contained in register AX to the memory portion of the stack. The subsequent "Push" in the

"FLOAT" sequence will create stack space for the 32 bit result. The concluding "Pop" will reload the top stack element in the AX register. Note that the INTEGER type rather than the CARDINAL type is used in the conversions.

8.3.1 Special NDP instructions

Apart from the basic arithmetic instructions the NDP has several special instructions for loading mathematical constants, to compute square roots, logarithms and transcendental functions, and to scale floating point numbers. It also has a number of processor control instructions and direct stack manipulating instructions.

The following example demonstrates how to place the floating point number of the mathematical constant "pi" on the top of the evaluation stack.

Example

```
SYSTEM PROCEDURE PI: REAL;
  (* A function to generate pi *)
BEGIN
  RULE <Expr> ::= PI ( ) DO

    Push(`AX`);
    Push(`AX`);
    Push(`AX`);
    EmitRR(  `MOV`,      `BX`, `SP`      );
    Emit(    `WAIT`      );
    Emit(    `FLDPI`      );
    Emit(    `WAIT`      );
    , EmitNDPA( `FSTP`, Real32, `SS: [BX]` );
    Pop(`AX`);

  END;
END;
```

The key instruction in the sequence above is the "FLDPI" instruction that loads (pushes) an 80 bit floating point representation onto the NDP stack. Corresponding instructions exist for other mathematical constants. The initial "Push" operations are required in order to create space for the resulting operand in the memory part of the CPU stack.

Another example demonstrates the computation of the square root of an

operand. The square root of any operand type supported by the NDP can be defined in this way.

Example

```
SYSTEM PROCEDURE SQRT(CARDINAL): CARDINAL;
    (* A function to compute the square root *)
BEGIN
    RULE <Expr> ::= SQRT ( <Expr> ) DO

        Push(`AX`);
        EmitRR( `MOV`,      `BX`, `SP`      );
        Emit(    `WAIT`                      );
        EmitNDPA( `FILD`, Int16, `SS: [BX]` );
        Emit(    `WAIT`                      );
        Emit(    `FSQRT`                     );
        Emit(    `WAIT`                      );
        EmitNDPA( `FISTP`, Int16, `SS: [BX]` );
        Pop(`AX`);

    END;
END;
```

The "FSQRT" instruction replaces the top stack operand in the NDP register stack with its square root.

8.3.2 Error handling

There are two general ways to handle errors in the NDP.

Internal error handling: The NDP will take default actions when an error condition is detected. The result may be erroneous. Each of the six error conditions can be individually set in a control word to indicate if default actions should be taken or not. The programmer must react to the result to detect that an error occurred.

External error handling: The NDP is connected to the interrupt logic in the system. The invoked interrupt handler (executed by the main processor) will inquire the NDP status and take necessary actions.

The former approach requires that error checking instructions are generated by the code rules, or alternatively that an error checking run-time procedure is called each time the NDP has been active. The latter case allows the error handling routines to be programmed as interrupt processes at the Modula-2 level.

CHAPTER 9. IMPLEMENTATION OF PROCESS CONTROL

The basic process control operations in a Modula-2 system are the NEWPROCESS, TRANSFER and IOTRANSFER operations.

The operation

NEWPROCESS(P,A,n,p)

will set up a workspace of size "n" at address "A" to be used by process "p". The process will start to execute procedure "P" at next TRANSFER (or IOTRANSFER) to "p". "P" must denote a parameterless procedure declared at top level.

The operation

TRANSFER(p1,p2)

will save the status of the current process and assign the current stack pointer to "p1". The stack segment pointer will be loaded with the contents of variable "p2", the status saved in that workspace will be loaded, and the execution path will be resumed.

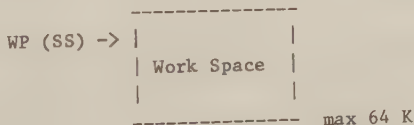
The operation

IOTRANSFER(p1,p2,va)

works as TRANSFER, but in addition causes an interrupt associated with vector entry "va" to assign the interrupted process to "p2" and then "p1" will be resumed. The IOTRANSFER implementation and other interrupt control operations will be discussed in chapter 10.

9.1 PROCESS CONTROL OPERATIONS IN THE iAPX86

In the iAPX86 implementation PROCESS objects are represented by 16 bit "Work Space Pointers" that will be loaded into the stack segment register when the process is running.



The use of a stack segment pointer to implement a variable of type PROCESS implies that a workspace must be allocated at a 16 byte boundary.

9.1.1 Implementation of NEWPROCESS

The NEWPROCESS operation may be implemented as a system procedure setting up a call to a central run-time procedure.

```
SYSTEM PROCEDURE NEWPROCESS(P:PROC,A:ADDRESS,n:CARDINAL,VAR p:PROCESS);
```

```
BEGIN
```

```
    RULE <SysProc> ::= NEWPROCESS(<Expr>,<Expr>,<Expr>,<Var>) DO
```

```
        EmitRI( 'MOV', 'SI',<VarP>.offset );
```

```
        GetDS('DX',<Var>);
```

```
        CallRTS('Newprocess');
```

```
    END
```

```
END;
```

At entry to the run-time procedure the "SI" register will contain the offset to the PROCESS variable and the "DX" register will contain the segment address. The values of the "P", "A", and "n" parameters will be pushed on the stack.

The task of the run-time part can be described as

```
RUNTIME PROCEDURE Newprocess;
```

```
BEGIN
```

```
    (*
```

```
    Newprocess:
```

```
        ; Set up an initial stack frame at
```

```
        ; segment address "A" with the size of "n"
```

```
        ; containing the information expected by
```

```
        ; the TRANSFER and IOTRANSFER operations.
```

```
        ; The resume address should be the entry of procedure "P".
```

```
        ; The workspace is assigned to process "p".
```

```
    *)
```

```
END;
```

It is reasonable to program a separate run-time part of the NEWPROCESS operation instead of generating straight code. The same is true for the other process handling operations. However, if execution time is crucial and the size of generated code is not, then the straight code alternative or any other division of the implementation alternatives may be more appropriate.

9.1.2 Implementation of TRANSFER

The TRANSFER operation may be implemented according to the following system procedure.

```
SYSTEM PROCEDURE TRANSFER(VAR p1: PROCESS, VAR p2: PROCESS);
BEGIN
    RULE <SysProc> ::= TRANSFER (<Var>,<Var>) DO
        Save;
        EmitRI(  ^MOV^      ^SI^,<Var>.offset  );
        GetDS(^AX^,<Var>);
        EmitRI(  ^MOV^      ^DI^,<Var>.offset  );
        GetDS(^BX^,<Var>);
        Emit(    ^PUSHF^                      );
        CallRTS(^Transfer^);
        Restore;
    END;
END;
```

The "Save" and "Restore" support procedures are supposed to be declared in the implementor module. They will generate the instructions to save and restore the contents of the working registers before and after the process transfer. The segment addresses and the offsets to the process parameters are contained in registers at entry to the run-time part. The "PUSHF" instruction will save the status of the running process in the current workspace.

RUNTIME PROCEDURE Transfer;

BEGIN

(*

Transfer:

CLI		;Disable Interrupts
PUSH	DS	;Save Data Segment Pointer (p1)
PUSH	ES	;Save Extra Segment Pointer (p1)
MOV	SS:.0,SP	;Save Stack Pointer (p1)
MOV	DS,AX	;Load DS (p1)
MOV	[SI],SS	;Save Work Space Pointer (p1)
MOV	DS,BX	;Load DS (p2)
MOV	SS,[DI]	;Load Work Space Pointer (p2)
MOV	SP,SS:.0	;Load Stack Pointer (p2)
POP	ES	;Load Extra Segment Pointer (p2)
POP	DS	;Load Data Segment Pointer (p2)
IRET		;Restore Status Word and Resume

*)

END;

The above implementation clearly demonstrates the very simple mechanism required to perform a transfer between two processes. There is no bookkeeping involved in the transfer operation except for the switch of registers and restoring of status.

9.2 PROCESS CONTROL OPERATIONS IN THE iAPX286

When executing in the protected virtual address mode the iAPX286 supports a process type called "task" that is tailored for use in large multi-user systems. Naturally, an operating system written in Modula-2 should be able to control and to fully utilize the processor support for such tasks. By a combination of tables declared at the programming level, system types, system procedures and run-time procedures it will be possible to obtain the desired degree of support for the iAPX286 task facilities. The following will give an idea of how this can be accomplished.

The task type itself is implemented by a 16 bit selector.

SYSTEM TYPE TASK = PROCESS OF 16

The task selector will point to the associated task state segment descriptor residing in a descriptor table. The internal task register in the CPU will contain the current task selector during execution.

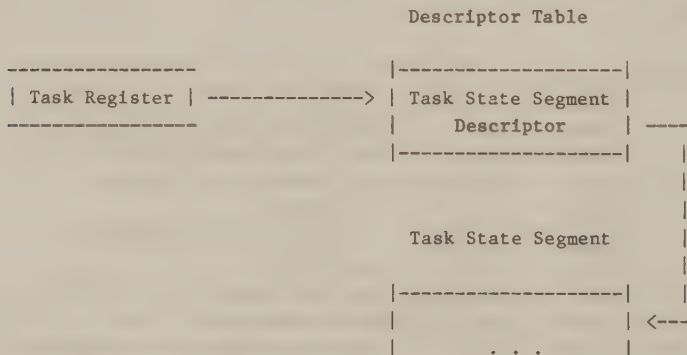


Fig. 9.1
Task State Segment References

The task state segment descriptor accommodates the address information and access rights to the task state segment containing the entire execution state for that task.

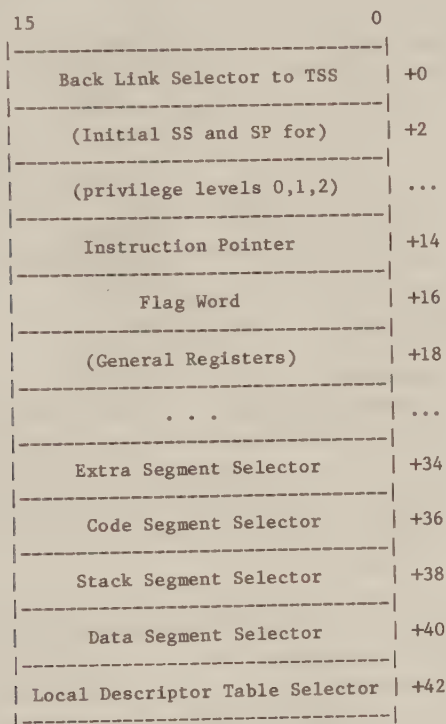


Fig. 9.2
Task State Segment

Within the task state segment all information regarding selectors, general registers and state information is contained. A task switch is performed when the processor running in protected mode executes an inter-segment JMP or CALL instruction which refers to a task state segment (or a task gate descriptor not discussed here).

The descriptor tables and the task state segments must be declared and initialized in the Modula-2 program before the processor can be switched over to the protected mode where the task support is available.

The following specification demonstrates the Modula-2 declarations for a task state segment.

TYPE

```

TaskStateSegment = RECORD
    BackLinkToTSS:      Selector;
    InitialCPLStacks:   ARRAY [0..2] OF
                        RECORD
                            SP: WORD;
                            SS: WORD;
                        END;
    InstructionPointer:  WORD;
    Flags:              FlagWord;
    GeneralRegisters:   ARRAY [RegSequence] OF WORD;
    ExtraSegment:       Selector;
    CodeSegment:        Selector;
    StackSegment:       Selector;
    DataSegment:        Selector;
    LocalDescriptorTable: Selector;
END;
```

The task state segment declaration will require the following support declarations.

```

Selector =      WORD RECORD
    RequestedPrivilegeLevel[BIT: 0..1]: PrivilegeLevelType;
    TableIndicator[BIT: 2]:             TableIndicatorType;
    Index[BIT: 3..15]:                  [0..8191];
END;
```

```

PrivilegeLevelType =
    (Kernel, SystemService, CustomExtensions, Applications);
```

```

TableIndicatorType =
    (UseGlobalDescriptorTable, UseLocalDescriptorTable);
```

```

RegSequence = (AX, CX, DX, BX, SP, BP, SI, DI);
```

Note especially the use of the BIT specifications. Such specifications are not standard Modula-2 but introduced for use in implementation dependent parts of this compiler system only (see section 4.1.2).

The advantages of the BIT notation is even more clearly demonstrated in the declaration of the iAPX286 flag word.

```
FlagWord =          WORD RECORD
    Carry[BIT: 0]:   BOOLEAN;
    Parity[BIT: 2]:   (None,Even);
    AuxiliaryCarry[BIT: 4]: (None,Borrow);
    Zero[BIT: 6]:     BOOLEAN;
    Sign[BIT: 7]:     (Positive,Negative);
    TrapFlag[BIT: 8]: (FALSE,SingleStep);
    InterruptEnable[BIT: 9]: BOOLEAN;
    DirectionFlag[BIT: 10]: (Increment,Decrement);
    OverflowFlag[BIT: 11]: BOOLEAN;
    IOPrivilegeLevel[BIT: 12..13]:
        (PrivilegeLevelType);
    NestedTaskFlag[BIT: 14]: BOOLEAN;
END;
```

In addition to the preparation of the task state segment for the main task the initiating program sequence should also load the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) base registers with references to valid tables. This will be further discussed in chapter 12 in connection with the protected virtual address mode.

9.2.1 Implementation of NEWTASK

A task is set up in correspondence to the NEWPROCESS operation in the iAPX86 using a selector "s" to the workspace instead of the segment address "A" and a TASK type parameter "t" instead of the PROCESS type parameter "p".

```
SYSTEM PROCEDURE NEWTASK(P: PROC,s: Selector,n: CARDINAL,VAR: t: TASK);
```

```
BEGIN
```

```
    RULE <SysProc> ::= NEWTASK(<Expr>,<Expr>,<Expr>,<Var>) DO
```

```
        GetDS('AX',<Var>);
```

```
        EmitRI('MOV', 'SI', <Var>.reladdr );
```

```
        CallRTS('NewTask');
```

```
    END;
```

```
END;
```


The run-time implementation of the "NewTask" procedure will set up a new task state segment according to fig. 9.2 .

RUNTIME PROCEDURE NewTask;

BEGIN

(*

```
; P - Task start procedure
; s - Selector to workspace descriptor
; n - Workspace size
; t - Selector to task state segment descriptor
```

NewTask:

```
;Disable Interrupts
;Save DS
;Load DS with AX value (DS of <Var>)
;Load SS (CPL 0) with workspace descriptor selector (s)
;Load SP (CPL 0) with workspace size (n)
;Load IP (Entry Point) with the offset part of PROC
;Clear Flag Word (interrupts off)
;Load ES selector with caller's ES
;Load CS selector with the CS part of PROC
;Load SS selector with workspace descriptor selector (s)
;Load DS selector with caller's DS
;Load Task LDT selector with caller's LDT
;Restore DS
;Enable Interrupts and Return
```

*)

END;

Before the NEWTASK procedure can be called the workspace descriptor (selected by "s") and the task state segment descriptor (selected by "t") must have been set up in either the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT). In addition the processor must run in protected virtual address mode.

The table setup and the processor mode transfer will be further discussed in chapter 12.

When executing in the protected virtual address mode the DS, CS, ES, and SS registers are supposed to contain selectors, i.e. pointers to descriptors, rather than the segment addresses used in the real address mode. This is a matter of interpretation of the register contents only and will not affect the code generated.

9.2.2 Implementation of TASKTRANSFER

Once the task state segments have been set up the transfer between tasks is handled entirely by the processor. It will be sufficient to implement the task transfer with a system procedure generating straight code with no reference to any central run-time procedure.

```
SYSTEM PROCEDURE TASKTRANSFER(VAR t1: TASK, VAR t2: TASK);
```

```
BEGIN
```

```
    RULE <SysProc> ::= TASKTRANSFER( <Var>,<Var> ) DO
```

```
        EmitR(  'PUSH',  'DS'                                );
```

```
        GetDS('DS',<Var.0>);
```

```
        EmitRM(  'STR',    <Var.0>.offset                    );
```

```
        EmitR(  'POP',    'DS'                                );
```

```
        GetDS('DS',<Var.1>);
```

```
        EmitRM(  'MOV',    'BX',<Var.1>.offset );
```

```
        EmitA(  'JMP',    '[BX]'                                );
```

```
    END;
```

```
END;
```

In the system procedure above the "t1" parameter will receive the contents of the internal task register, i.e. a selector to the task state segment descriptor defining the current task. The task transfer will take place when the processor detects that the destination of the "JMP" instruction is defined by a selector to a task state segment descriptor.

The task handling facilities in the iAPX86 is an example of extensive hardware support requiring only a thin layer of low-level control to be available for use directly from the Modula-2 level. The approach taken by the adaptable compiler system integrates the low-level control in the code generated in contrary to systems relying on separately translated assembler routines.

CHAPTER 10. IMPLEMENTATION OF INTERRUPT CONTROL

In terms of Modula-2, an interrupt is a process transfer operation. The transfer to take place when a certain interrupt occurs is registered by a call to

```
PROCEDURE IOTRANSFER(VAR p1,p2: PROCESS; va: CARDINAL)
```

The IOTRANSFER operation suspends the calling (device) process, assigns it to "p1", resumes the suspended process "p2", and in addition causes the interrupt transfer occurring at a later point to assign the interrupted process to "p2" and resume the device process "p1". The interrupt vector address assigned to the device is given by "va".

In the PDP-11 implementation the IOTRANSFER operation should only be used within modules with a specified interrupt priority level.

Example

```
MODULE Typewriter[4];      (* Interrupt priority level = 4 *)

  FROM SYSTEM IMPORT
    IOTRANSFER,PROCESS,...

  ...
  IOTRANSFER(p1,p2,va);
  ...

END Typewriter;
```

With this setup the execution of a program can be interrupted if, and only if, the interrupting device has a priority greater than the priority level of the module containing the statement currently being executed. If an explicit specification is absent, the priority level in any procedure is that of the calling procedure [Wirt82].

10.1 INTERRUPT CATEGORIES

Interrupts can be caused by external signals or by internal conditions or instructions. Interrupts can further be maskable or non-maskable

and, depending on the processor, maskable interrupts can be outmasked individually or on a group basis.

Internal non-maskable interrupts must always be taken care of by any stand-alone Modula-2 program, if such interrupts may occur as a result of "normal" program execution. The "Division By Zero" interrupt in the iAPX86 processor is an example of this kind of interrupt.

Internal software controlled interrupts must be taken care of only if they are used by the implementation. These interrupts will occur as a result of the testing for a certain condition, such as an arithmetic overflow followed by the execution of a special instruction that will cause the interrupt if the condition is true.

External non-maskable interrupts may occur only if the actual configuration is designed in that way. A typical use would be to activate a power failure routine.

External maskable interrupts can be disabled internally by software, i.e. the processor will in that case not respond to that type of interrupt until enabled.

Vectored interrupts implies that interrupting devices has a means of identifying themselves in order to be handled by dedicated interrupt service routines registered in a vector table.

10.2 IMPLEMENTATION OF IOTRANSFER

The objectives for the implementation of the Modula-2 IOTRANSFER operation in various micro system configurations depends on the main processors interrupt facilities and in some cases also on the characteristics of other interrupt controlling components in the family. In order to make process transfers efficient the IOTRANSFER and TRANSFER operations should be implemented in close agreement with the hardware supported interrupt mechanism. The process status saved in the workspace should if possible be organized in a uniform way allowing the TRANSFER/IOTRANSFER operations to be applied in any order without any time consuming case tests.

In the following the basic interrupt facilities in the iAPX86 processor will be presented to provide a frame for the control of low-level interrupt operations from the Modula-2 level.

Type 0. Divide by zero interrupt. (Non-maskable internal interrupt)

An attempt to divide by 0 will cause an interrupt via vector entry 0. This interrupt must be taken care of by any Modula-2 implementation that does not perform an explicit zero check of the dividend operand of a division instruction at run-time.

Type 1. Single step interrupt. (Non-maskable internal interrupt)

The processor can be set to execute in single-step mode causing an interrupt via vector entry 1 to occur after each instruction.

This facility is primarily intended for debugging at assembler level.

Type 2. External non-maskable interrupt

A non-maskable interrupt will always force the processor to respond via vector entry 2 independent of the setting of the interrupt status bit.

The non-maskable interrupt handling in Modula-2 is programmed and processed in the same way as the other external interrupts. The occurrence of a non-maskable interrupt within an outmasked critical code sequence might invalidate further execution. Non-maskable interrupts are recommended to be used only in exceptional situations.

Type 3. Software interrupt. (Non-maskable internal interrupt)

The programmed execution of the Interrupt instruction (INT) will cause an interrupt via the vector entry specified in the operand. In the case the operand is missing vector entry 3 will be used.

The general form can be used to implement system calls, signals, exceptions and other interrupt driven communication operations. The latter form is especially useful for setting breakpoints when programs are debugged as the INT instruction occupies one byte only.

Type 4. Overflow interrupt. (Non-maskable internal interrupt)

The programmed execution of an Interrupt-On-Overflow instruction (INTO) will cause an interrupt via vector entry 4 if the Overflow flag

in the processor status word is set.

This interrupt type is of interest to use primarily at the implementation level.

Type 5-255. Maskable external interrupts.

The user defined maskable external interrupts are registered in vector entries 5 up to 255. When a maskable interrupt occurs the vector entry number is obtained by the processor from external interrupt logic, usually a programmable interrupt controller.

Interrupt Vector Table

Type 0	Divide by zero interrupt
Type 1	Single step interrupt
Type 2	Non-maskable interrupt
Type 3	Software interrupt
Type 4	Overflow interrupt
Type 5	User defined interrupt
...	...
Type 255	User defined interrupt

Each interrupt vector entry occupies four bytes.

IP	Instruction Pointer
CS	Code Segment Pointer

The interrupt vector table will be assumed to be located at absolute address 0 to 1023 ($=256 \times 4 - 1$) and will not require an explicit declaration.

An interrupt vector entry is initiated as a result of an IOTRANSFER operation. It will contain a pointer to the interrupt entry where the execution of the registered interrupt process should start. For each interrupt type possible to occur the corresponding interrupt process entry must be registered in the vector table.

The initial registration of an interrupt process is exemplified by the "DivideByZeroProcess".

Example

```

VAR MainProcess,
    InterruptedProcess,
    DivideByZeroProcess:    PROCESS;

    Workspace:    ARRAY [0..30] OF WORD;
                . . .

PROCEDURE DivideByZeroInterrupt;
BEGIN
    LOOP
        IOTRANSFER(DivideByZeroProcess, InterruptedProcess, 0);
        < Take action >
    END
END DivideByZeroInterrupt;
    . . .

```

In the initiation sequence in the main program:

```

NEWPROCESS(DivideByZeroInterrupt, Workspace, SIZE(Workspace),
           DivideByZeroProcess);
TRANSFER(InterruptedProcess, DivideByZeroProcess);
    . . .

```

When a divide by zero interrupt occurs the interrupted process will be assigned to the process variable "InterruptedProcess" and thus be resumed when the "DivideByZeroProcess" has taken action.

The dual purpose of the second process parameter to the IOTRANSFER operation can be confusing in some applications. It specifies both the process to be immediately transferred to as well as the process variable to receive the assignment of the interrupted process. A third process parameter would make it possible to separate the transfer

process from the interrupted process. However, it is the implementation of the original suggestion of the IOTRANSFER operation we will discuss here.

Implementation of the IOTRANSFER operation

The following IOTRANSFER operation is implemented as a system procedure interacting with two run-time procedures.

```

SYSTEM PROCEDURE IOTRANSFER(VAR p1: PROCESS, VAR p2: PROCESS,
                             va: CARDINAL);
BEGIN
    RULE <SysProc> ::= IOTRANSFER (<Var.0><Var.1><Expr>) DO

        GetDS('DX', <Var.0>);                                (* 1 *)
        EmitRI('MOV', 'SI', <Var.0>.offset);                  (* 2 *)
        GetDS('CX', <Var.1>);                                (* 3 *)
        EmitRI('MOV', 'DI', <Var.1>.offset);                  (* 4 *)
        CallRTS('IoTransfer');                                (* 5 *)
        MakeLabel(<Var.0>.lab);                               (* 6 *)
        EmitA('JMP', <Var.0>.lab);                            (* 7 *)
        (* Irpt Entry *)                                      (* 8 *)
        EmitR('PUSH', 'AX');                                  (* 9 *)
        EmitR('PUSH', 'CX');                                  (* 10 *)
        EmitR('PUSH', 'DX');                                  (* 11 *)
        EmitR('PUSH', 'BX');                                  (* 12 *)
        GetDS('AX', <Var.0>);                                  (* 13 *)
        EmitRI('MOV', 'DX', <Var.0>.offset);                  (* 14 *)
        GetDS('CX', <Var.1>);                                  (* 15 *)
        EmitRI('MOV', 'BX', <Var.1>.offset);                  (* 16 *)
        CallRTS('IrptTransfer');                              (* 17 *)
        EmitLabel(<Var.0>.lab);                                (* 18 *)

    END
END;
```

The instruction (* 1 *) loads the DX register with the segment address to the first process variable. Instruction (* 2 *) moves the offset address to the process variable into the source index register (SI). Instruction (* 3 and 4 *) loads the segment and offset addresses of the second process variable into the CX and DI register respectively. In instruction (* 5 *) the run-time procedure "IoTransfer" is invoked.

The "IoTransfer" procedure will register the first parameter as the interrupt process and perform the transfer to the process specified by

the second parameter.

The second section of the system procedure defines the interrupt entry part, i.e. the sequence executed in response to an interrupt via vector entry "va". The instructions (* 9 to 12 *) will save the contents of some of the registers in the stack in order to make the registers available for the subsequent loading of the variable parameters (* 13 to 16 *) before the "IrptTransfer" procedure is entered (* 17 *).

The "IrptTransfer" procedure will preserve the execution status of the interrupted process and assign it to the second process parameter, then the process saved by the first section will be resumed. When this process starts execution it will enter the code at (* 6 *) and thereby perform a bypass jump to the label emitted by (* 18 *).

10.3 PRIORITY INTERRUPTS

The PDP-11 implementation of Modula-2 suggests that the interrupt priority should be specified by a constant parameter in the module head. The execution within a module with a given priority level may be interrupted only by devices given a higher priority level. This scheme will force the interrupt processing to conform to the hierarchical arrangement defined by the interrupt priority parameters, thereby enforcing a restrictive but safe frame for the interrupt processing together with a visually clear notation.

The implementation of the suggested priority scheme requires that priority controlling instructions are generated at all entries and exits within each module having a specified interrupt priority. On entry the current interrupt priority must be saved at run-time before the new is set. On exit the saved priority should be reloaded. In addition the interrupt priority of a process must be saved and reloaded on all process transfers including interrupts.

In order to support the suggested scheme the front end part of the compiler will generate calls in intermediate representation to the system procedures SETPRIORITY and RESETPRIORITY on each entry and exit of a module with the interrupt priority specified in the module head. The corresponding code generation rules must be supplied in a component module.

```
RULE <SysProc> ::= SETPRIORITY ( <Constant> ) DO
```

```
. . .
```

```
END;
```

```
RULE <SysProc> ::= RESETPRIORITY ( ) DO
```

```
. . .
```

```
END;
```

The saving and reloading of the interrupt priority associated with the process transfer, iotransfer and interrupt transfer operations is accomplished by adding the appropriate instructions to the implementing code for each of these operations. Some processors, such as the MC68000, automatically saves the priority level as a part of the processor status word when an interrupt occurs.

Most micro processors rely on external interrupt prioritizing components. Such components may be programmed and controlled directly from the Modula-2 level allowing the interrupt processing to be organized under total control of the programmer. The SETPRIORITY/RESETPRIORITY scheme is a convenient way to implicitly invoke the actual priority mechanism with the module head constant as a parameter.

Instructions for enabling and disabling interrupts can be generated by calls to the system procedures IRPTENABLE and IRPTDISABLE respectively.

Examples

```
SYSTEM PROCEDURE IRPTENABLE;
```

```
RULE <SysProc> ::= IRPTENABLE () DO
```

```
    Emit(      'STI'      );
```

```
END;
```

```
SYSTEM PROCEDURE IRPTDISABLE;
```

```
RULE <SysProc> ::= IRPTDISABLE () DO
```

```
    Emit(      'CLI'      );
```

```
END;
```

CHAPTER 11. MULTI-PROCESSOR SYSTEM SUPPORT

Multi-processor systems imply that processes may be executed concurrently by separate processors. In such systems it is most desirable that the concurrency can be described in conceptually clear terms which can be implemented at the software level and supported by the hardware. The Modula-2 PROCESS type together with the TRANSFER and the IOTRANSFER operations are facilities on a very basic level that are intended as primitives to construct multi-programming mechanisms tailored to the actual application. In this chapter the hardware support for concurrent processes in multi-processor configurations will be discussed.

Critical regions and semaphores

The concept of critical region is of primary importance when programming systems of interacting concurrent processes. The idea is to control a section of code by a mechanism that grants that only one process at a time may enter. In this way sequential paths in the program will be established where the programmer will get a chance to cope with the concurrency. Entrance and exit to a critical region are controlled by a semaphore. A process wishing to enter examines the semaphore. If it indicates that no other process is executing within the critical region, the process may enter and at the same time change the semaphore to indicate the presence of a process within the critical region. When the process exits it resets the semaphore. If a process tries to enter a critical region when the semaphore indicates the presence of another process, the process must wait until the semaphore is reset.

The implementation of a critical region system must solve the following problems.

- 1) What happens if two processes examine the semaphore at the same time?
- 2) How should a process wait if it is not allowed to enter?
- 3) How should a waiting process be resumed when the semaphore is reset?

The first problem may be solved by a so called indivisible operation,

i.e. an operation that can not be performed by two processes at the same time. Indivisible operations may be implemented by the aid of a "test and set instruction" that reads a flag and sets it within the execution of a not interruptable instruction. The flag will be zero only to the first process executing the "test and set instruction". The setting of the flag will stop other processes to proceed while the first process executes the critical region. The problem of implementing a critical region of code is thus solved by a "critical instruction" supported by the hardware.

The second problem may be solved by either so called "busy waiting" implying that the waiting process repeatedly tests the entry condition to the critical region until it becomes true, or it may be solved by a "wait" operation that suspends the process until the condition becomes true.

The third problem may be solved by an "exit" or "signal" operation performed by a process leaving the critical section.

11.1 IMPLEMENTATION OF CRITICAL REGIONS

The support to implement operations used to establish critical regions in multi-processor configurations differs among the processor families. The following will give a brief presentation of the basic support in a few families and how the instructions can be tied to system procedures at the Modula-2 level.

11.1.1 The iAPX86 family

The iAPX86 instruction set does not include a pure "test and set instruction" but provides a bus lock prefix that may be used to implement an instruction sequence with similar effect. A critical region with a busy wait loop may be established as follows at the assembly level.

```
Wait:
      MOVI     AL,1      ;Set "busy" indicator
LOCK  XCHG     AL,flag   ;Swap (AL) and (flag)
      TEST    AL,AL     ;Test value of previous flag
      JNZ     Wait      ;Jump to "Wait" if "busy"
      < Critical region >
      MOVI     flag,0    ;Reset flag to "free"
```

This sequence corresponds to the following "Enter" and "Exit" system procedures.

```
SYSTEM PROCEDURE Enter(VAR: BYTE);
```

```
BEGIN
```

```
    RULE <SysProc> ::= Enter( <Var> ) DO
```

```
        MakeLabel(<Var>.lab);
        EmitLabel(<Var>.lab);
        EmitRI(  'MOVI',  'AL',1          );
        Emit(    'LOCK'   );
        EmitRM(  'XCHG',  'AL',<Var>      );
        EmitRR(  'TEST',  'AL','AL'      );
        EmitA(   'JNZ',   lab             );
```

```
    END;
```

```
END;
```

```
SYSTEM PROCEDURE Exit(VAR: BYTE);
```

```
BEGIN
```

```
    RULE <SysProc> ::= Exit( <Var> ) DO
```

```
        EmitM(  'MOVI',  <Var>,0         );
```

```
    END;
```

```
END;
```

A "LOCK" prefix generated as a separate instruction might be misplaced if the succeeding instruction, i.e. the instruction to be prefixed, contains a variable requiring extra instructions to be referenced. In such cases a special code emitting procedure must be programmed and included in the implementor module.

12.1.2 The Z8000 family

The Z8000 family relies on handshaking signals to control access to shared resources. The MI signal is used to find out if a shared resource, such as a memory, is available. The MO output signal is used to perform a shared resource request. The four instructions MBIT, MREQ, MRES, and MSET manipulates the two handshaking signals.

With the aid of these instructions the critical region entry and

exit can be programmed as system procedures.

SYSTEM PROCEDURE Enter;

BEGIN

RULE <SysProc> ::= Enter DO

```
Emit(      ^MSET^          ); (* Request resource *)
MakeLabel(<SysProc>.lab);
EmitLabel(<SysProc>.lab);      (* Wait#:      *)
Emit(      ^MBIT^          ); (* Test if free *)
EmitFA(     ^JP^    ^PL^,<SysProc>.lab ); (* Jump until free *)
```

END;

END;

SYSTEM PROCEDURE Exit;

BEGIN

RULE <SysProc> ::= Exit DO

```
Emit(      ^MRES^          ); (* Release resource *)
```

END;

END;

11.1.3 The M68000 family

The MC68000 family includes a proper "test and set" instruction (TAS) in the instruction set. The high-order bit (7) of a memory byte will be set while the previous setting of that bit affects the status bits (Zero and Negative).

SYSTEM PROCEDURE Enter(VAR: BYTE);

BEGIN

RULE <SysProc> ::= Enter(<Var>) DO

```
MakeLabel(<Var>.lab);
EmitLabel(<Var>.lab);      (* Wait#:      *)
EmitM(     ^TAS^,    <Var>    ); (* Test and set bit 7 *)
EmitA(     ^BNE^,    <Var>.lab ); (* Branch if set (negative) *)
```

END;

END;


```
SYSTEM PROCEDURE Exit(VAR: BYTE);
```

```
BEGIN
```

```
    RULE <SysProc> ::= Exit( <Var> ) DO
```

```
        EmitMI(  "MOVEB", <Var>,0 );  (* Clear byte *)
```

```
    END;
```

```
END;
```

11.2 IMPLEMENTATION OF SEMAPHORES

The busy wait scheme is simple and fast but is clearly a waste of the processor resource if the system contains other executable processes. In this case a "semaphore" construct is more appropriate to control the entry to (and exit from) a critical region.

A semaphore is composed of a non-negative counter and a queue. The semaphore may only be operated by dedicated operations such as "Initiate", "Wait" and "Signal". The counter indicates the number of processes which may pass the semaphore without being suspended when executing the "Wait" operation. When used for a critical region the counter is initially 1 and is set to 0 when the critical region is occupied. The queue will record the processes suspended by the "Wait" operation and thus waiting to pass the semaphore.

In terms of Modula-2 a semaphore and the associated operations can be implemented by the following routines using the "Enter" and "Exit" operations.

```
TYPE
```

```
    semaphore = RECORD
```

```
        flag:    BYTE;
```

```
        count:   INTEGERB;
```

```
        queue:   POINTER TO PROCESS
```

```
    END;
```

```
PROCEDURE Initiate(VAR sem: semaphore; init: INTEGERB);
```

```
BEGIN
```

```
    sem.count:=init;    (* =1 for critical region *)
```

```
    sem.queue:=NIL;
```

```
END Initiate;
```

```

PROCEDURE Wait(VAR sem: semaphore);
BEGIN
    Enter(MultiProtect);
    DEC(sem.count);
    IF sem.count < 0 THEN
        < Enter calling process in sem.queue >
        < Start another process >
    END;
    Exit(MultiProtect);
END Wait;

PROCEDURE Signal(VAR sem: semaphore);
BEGIN
    Enter(MultiProtect);
    INC(sem.count);
    IF sem.count >= 0 THEN
        < Remove a process from sem.queue >
        < Make it ready to run >
    END;
    Exit(MultiProtect);
END Signal;

```

The interaction between concurrent processes in a multi-processor system can become very complicated. The degree of complication is often dependent on the demands for efficiency. In such situations the conflict between the clarity of a high-level language and the low-level control of the assembly language is more evident than ever. The system procedures supported by the adaptable compiler system solves this conflict and allow the programmer to integrate the two levels.

CHAPTER 12. PROTECTION AND VIRTUAL MANAGEMENT CONTROL

Protection and virtual management control are facilities of interest to control in multi-user or multi-tasking applications. The protection control may be aimed at use of privileged instructions or it may control the access to certain memory areas. The virtual management control concerns the implementation of logical address spaces. Protection and virtual memory management are often implemented in close connection using the same set of tables and supporting hardware mechanisms.

In this chapter the protection and virtual management facilities present in the Z8000 and IAPX286 families will be discussed.

12.1 SIMPLE PROTECTION MECHANISMS

A simple form of protection control is a hardware support for system/user execution. In the user mode certain instructions may not be executed, and certain memory areas may not be accessed. The transfer from system mode to user mode is performed by a privileged instruction and the reverse by a system call instruction or an illegal operation causing a violation interrupt.

The example below illustrates the specification of the system call instruction in the Z8000.

Example

```
SYSTEM PROCEDURE SystemCall(Function: BYTE);  
  
BEGIN  
    RULE <SysProc> ::= SystemCall( <Constant> ) DO  
        EmitI(    'SC',    <Constant>    );  
    END;  
END
```

The reversed function, i.e. the transfer to execution in the Z8000 protected mode can be performed either by setting the mode select bit

in the status control word, or by an interrupt return instruction.

Example

```
SYSTEM PROCEDURE SetUserStatus;

BEGIN
    RULE <SysProc> ::= SetUserStatus DO

        EmitRR(  'LDCTL',  'R1', 'FCW'    );
        EmitRI(  'AND',    'R1', '0BFFFH' );
        EmitRR(  'LDCTL',  'FCW', 'R1'    );

    END;
END;
```

The normal way to return to the protected mode in an executing Z8000 system is by an interrupt return instruction that will reset the control word to the contents at the point of the system call.

Example

```
SYSTEM PROCEDURE ExecuteUser;

BEGIN
    RULE <SysProc> ::= ExecuteUser DO

        Emit(      'IRET'    );

    END;
END;
```

12.2 PROTECTION AND MEMORY MANAGEMENT

12.2.1 The Z8010 Memory Management Unit

An example of an integrated approach to protection and memory management is found in the Z8010 memory management unit belonging to the Z8000 processor family.

The Z8010 Memory management unit (MMU) is configured in a special I/O space (SPIO) requiring dedicated I/O instructions. The MMU is controlled by reading/writing values from/to internal tables accessed via I/O ports. The I/O port addressing is somewhat unusual; the low order byte of the 16 bit I/O port number contains the MMU address while the high order byte is interpreted as a command code.

The nature of the MMU commands can be shown by a table.

Type	Command	Register accessed	Read/Write
C	00	Mode Control	R/W
C	01	Segment Descriptor Address	R/W
S	02	Violation Type	R
S	03	Violation Segment Number	R
	...		
D	08	Segment Descriptor Base Address	R/W
D	09	Segment Descriptor Limit Byte	R/W
	...		
C	20	Descriptor Selection Counter	

Type C = Control Register Access Command

D = Data Register Access Command

S = Status Register Access Command

To access the MMU registers the corresponding system procedures can be specified using the code emitting procedures defined in the implementor module for the Z8000 family.

```
SYSTEM PROCEDURE ReadViolationType(VAR ViolationReg: BYTE,
                                     Unit: BYTE);
```

```
BEGIN
```

```
    RULE <SysProc> ::= ReadViolationType(<Var><Constant>) DO
```

```
        EmitIR(  'SINB',    'R0',0100H+<Constant>    );
```

```
        EmitMR(  'LDB',     <Var>.offset,'RLO'        );
```

```
    END;
```

```
    RULE <SysProc> ::= ReadViolationType(<Var><Expr>) DO
```

```
        EmitIR(  'POP',     'R1'                        );
```

```
        EmitRI(  'LDB',     'RH1',01H                  );
```

```
        EmitRI(  'LD',      'R3',1                      );
```

```
        EmitRM(  'LDA',     'R2',<Var>.offset           );
```

```
        EmitRRR(  'SINIB',  '@R2','@R1','R3'            );
```

```
    END;
```

```
END;
```

The system procedure rules for the "ReadViolationType" operation show two ways to form the combined command-unit address. In the first code rule the unit is defined by a constant parameter thus allowing that a direct address is formed during compile time. In the second code rule

the unit is defined by an expression evaluated during run-time and pushed onto the stack. The "SIN(B)" instruction allows direct addresses only, why the increment version of the instruction (SINIB) is used instead. This instruction allows the destination operand to be referenced indirectly by a register, in this case the R2 register. The R3 register is used as a dummy counter register.

An alternative approach is to define the special I/O space by a system space declaration in the component module.

```
SYSTEM SPACE SPIO = [0..4];
```

The front end will now accept absolute allocation of variables using the imported prefix SPIO followed by a constant within the declared range.

Example

```
VAR MMU0[SPIO: 0]: BYTE;
    MMU1[SPIO: 1]: BYTE;
```

```
SYSTEM PROCEDURE ReadMMU(VAR Result: BYTE;
                          VAR MMU: BYTE, Command: BYTE);
```

```
BEGIN
```

```
    RULE <SysProc> ::= ReadMMU( <Var.0> <Var.1> <Constant> ) DO
```

```
        IF <Var.1>.absolute AND <Var.1>.prefix = SPIO
```

```
        THEN
```

```
            EmitRI(  "SINB",  "R0",
                    <Constant>.value*100H+<Var.1>.absaddr),
            EmitMR(  "LDB",    <Var.0>,"RLO"  );
```

```
        ELSE ,
```

```
            Error(...)
```

```
        END;
```

```
    END
```

```
END;
```

12.2.2 Protected Virtual Address Mode in the iAPX286

The iAPX286 also integrates the address space protection with the virtual memory support. After power-up it always executes in so called real address mode with all protection and memory management facilities disabled. Before the transfer to the protected virtual address mode

can take place descriptor tables must be set up. The "Global Descriptor Table" (GDT) base register and the "Interrupt Descriptor Table" (IDT) base register must be prepared to refer to valid entries in the descriptor tables. After the preparations the processor is ready for the transfer that will occur as a result of the execution of an instruction to set the protection enable status bit immediately followed by an intra-segment jump instruction in order to clear the internal pre-fetch queue from instructions decoded in real address mode.

In the virtual protected mode all memory references take place via the descriptor tables. A global descriptor table contains descriptors available to all tasks. Local descriptor tables contain descriptors that can be private to a task. A descriptor specifies the segment base address and other segment attributes such as segment size (max 64 k), access rights, presence in memory and information if the segment has been accessed. References to a segment not present in memory causes an internal exception with the state preserved upon return.

TYPE

```

DescrType = (CodeDescr,DataDescr);

CodeDataDescriptor = RECORD
    Limit:          CARDINAL;
    LowBase:        CARDINAL;
    HighBase:       CARDINALB;
    AccessRights:   BYTE RECORD
        A[BIT 0]:   (NotAccessed,Accessed);
        CASE DescrType OF
            CodeDescr:
                Readable[BIT 1]:      BOOLEAN;
                Conforming[BIT 2]:     BOOLEAN;
                Executable[BIT 3]:     BOOLEAN |
            DataDescr:
                Writeable[BIT 1]:      BOOLEAN;
                ExpansionDown[BIT 2]:  BOOLEAN;
                Executable[BIT 3]:     BOOLEAN
        END;
    SegDescr[BIT 4]:  BOOLEAN;
    DescrPrivLevel[BIT 5..6]: [0..3];
    Present[BIT 7]:   BOOLEAN
    END;
Reserved:           WORD
END;
```



```
SystemDescriptor = RECORD
```

```

    Limit:          CARDINAL;
    LowBase:        CARDINAL;
    HighBase:       CARDINAL;
    AccessByte:     BYTE RECORD
        TypeField[BIT 0..3]:      [1..3];
        DescrPrivLevel[BIT 5..6]: [0..3]
        ValidContent[BIT 7]:      BOOLEAN
    END;
    Reserved:       WORD
    END;
```

```
GateDescriptor = RECORD
```

```

    DestOffset:     CARDINAL;
    SecondWord:     WORD RECORD
        DestSelector[BIT 2..15]:  [0..16383]
    END;
    CountField:     BYTE RECORD
        WordCount[BIT 0..4]:      [0..31]
    END;
    AccessByte:     BYTE RECORD
        TypeField[BIT 0..3]:      GateType;
        DescrPrivLevel[BIT 5..6]: [0..3];
        ValidContent[BIT 7]:      BOOLEAN
    END;
    Reserved:       WORD
    END;
```

The loading of the internal LDT and GDT registers will provide the CPU with the base and limit values for the proper descriptor tables.

```
TYPE Descriptor = RECORD
```

```

    Limit:          CARDINAL;
    LowBase:        CARDINAL;
    HighBase:       CARDINAL;
    END;
```

```
SYSTEM PROCEDURE LoadGlobalDescrTableReg(VAR d: Descriptor);
```

```
BEGIN
```

```
    RULE <SysProc> ::= LoadGlobalDescrTableRegister( <Var> ) DO
```

```
        EmitM( 'LGDT', <Var> );
```

```
    END;
```

```
END;
```

The LGDT instruction will load the internal global descriptor base register with the limit and base values to the proper descriptor table.

TYPE

```

Selector =      WORD RECORD
    ReqPrivLevel[BIT 0..1]:  [0..3];
    TableIndicator[BIT 2]:   (UseGlobal, UseLocal),
    Index[BIT 3..15]:        [0..8191]
END;
```

SYSTEM PROCEDURE LoadLocalDescrTableReg(VAR s: Selector);

BEGIN

```

    RULE <SysProc> ::= LoadLocalDescrTableRegister( <Var> ) DO
```

```

        EmitM(    ^LLDT^,    <Var>    );
```

```

    END;
```

END;

The LLDT instruction will load the internal local descriptor table register with a selector referring to a local descriptor table.

The internal interrupt descriptor table register is loaded with a special instruction corresponding to the global descriptor table register.

SYSTEM PROCEDURE LoadInterruptDescrTableRegister(VAR d: Descriptor);

BEGIN

```

    RULE <SysProc> ::= LoadInterruptDescrTableRegister( <Var> ) DO
```

```

        EmitM(    ^LIDT^,    <Var>    );
```

```

    END;
```

END;

Next the Protection Enable Flag in the Machine Status word should be set by loading the status word with the proper bits set.

TYPE

```
StatusWord =  
    WORD RECORD  
        ProtectionEnable[BIT: 0]:    BOOLEAN;  
        MonitorProcessorExtension[BIT: 1]: BOOLEAN;  
        ProcessorExtensionEmulated[BIT: 2]:BOOLEAN;  
        TaskSwitch[BIT: 3]:          BOOLEAN;  
    END;
```

SYSTEM PROCEDURE LoadMachineStatusWordRegister(VAR w: WORD);

BEGIN

RULE <SysProc> ::= LoadMachineStatusWordRegister(<Var>) DO

EmitM('LMSW', <Var>);

END;

END;

After the LMSW instruction has set the ProtectionEnable bit the processor must immediately execute an intra-segment JMP instruction in order to clear the instruction pre-fetch queue from instructions decoded in real address mode.

SYSTEM PROCEDURE ClearQueue;

BEGIN

RULE <SysProc> ::= ClearQueue DO

EmitA('JMP', '\$+1');

END;

END;

Next, to force the iAPX286 to load the CPU registers with the initial values assumed by the program at start of the protected mode, another JMP instruction with a selector referring to the main task state segment should be performed. In addition the Task State register should be set to point at a valid task state segment as the inter-segment JMP instruction will cause a task switch operation to occur

which involves saving the current task state in that task state segment.

```
SYSTEM PROCEDURE StartExecution(VAR OldTSS,MainTSS: Selector);
```

```
BEGIN
```

```
    RULE <SysProc> ::= StartExecution( <Var> <Var> ) DO
```

```
        EmitM(  ^LTR^,      <Var.0>  )
```

```
        EmitM(  ^JMPS^,    <Var.1>  );
```

```
    END;
```

```
END;
```

Once running in the protected mode the processor will run in that mode until reset.

CHAPTER 13. UTILIZATION OF AN OPERATING SYSTEM COMPONENT

Firmware components is the suggested name for those components that implement special operations encoded in non-volatile (ROM) memory. They differ from the coprocessor components in the respect that the operations are implemented by instructions to be executed by the main processor itself rather by a built-in processor. The primary idea is that commonly used operations such as primitives in a real-time system should be encoded in a separate component where they are safe against unintentional alteration. The firmware component also offers a way of distributing efficiently coded and thoroughly tested operations on a mass basis. Optionally the firmware component might be integrated with hardware facilities supporting the implemented operations.

In the following the utilization of the Intel 80130 Operating System Firmware component (OSF) will be briefly discussed.

The OSF implements 35 primitive functions to be used by an accompanying main processor, i.e. an 8086/8088. It also contains an interrupt controller, a programmable interval timer and a baud rate generator. The control store comprises 16 k bytes. The primitive functions implement multitasking, interrupt management, free memory management, intertask communication and synchronization.

A primitive is invoked by the main processor following a standardized calling convention.

```
PUSH    P1        ; Push parameter 1
PUSH    P2        ; Push parameter 2
...
PUSH    Pn        ; Push parameter n

PUSH    BP        ; Save BP
MOV     BP,SP     ; BP:=(SP)

LEA     SI,SS: ParamSize+2 [BP] ; SI points to first
                                ; parameter

MOV     AX, EntryCode        ; Primitive Entry Code
INT     184                  ; Invoke OSF interrupt

< Execute OSF primitive >
...
POP     BP        ; Restore BP
RET     ParamSize ; Return and reset SP
```

Upon return from the OSF the registers will contain the following information:

CX	Exception codes
DL	The parameter number that caused the exception (if any)
AX	Word return value
ES: BX	Pointer return value

For each firmware implemented primitive of interest to use from the Modula-2 level a SYSTEM procedure may be defined.

Example

```
SYSTEM PROCEDURE CreateMailBox(MailBoxFlags,ExceptionPtr);
```

```
BEGIN
```

```
  RULE <SysProc> ::= CreateMailBox(<Expr><Expr>) DO
```

```
    EmitR(  ^PUSH^,  ^BP^                );
    EmitRR( ^MOV^,    ^BP^,^SP^           );
    EmitRA( ^LEA^,    ^SI^,^SS: .8[BP]    );
    EmitRI( ^MOV^,    ^AX^,300H           );
    EmitA(  ^INT^    ^184^                );
    (* Execute OSF primitive *)
    EmitR(  ^POP^,    ^BP^                );
    EmitRI( ^CMP^,    ^CX^,0              );
    EmitA(  ^JZ^,     ^$+2^                );
    EmitA(  ^INT^,    ^31^                );
    EmitA(  ^RET^,    ^6^                  );
```

```
  END;
```

```
END;
```

For each OSF primitive the parameters must be passed in accordance with the description for that primitive. The firmware procedure is invoked on an interrupt basis using entry 184 in the interrupt vector. Upon return the CX register may indicate that an exception occurred requiring a transfer to an exception handling routine. In the example above the exception will be handled by an interrupt process at the Modula-2 level programmed to respond to an interrupt via vector entry 31.

Compared to the rather complicated link scheme often necessary when using firmware components in a conventional development system, the procedure specifications nicely establish a direct binding with full control of the processor/firmware interface.

CHAPTER 14. CONCLUSIONS

14.1 SUMMARY

The major objective of this thesis was to provide a high-level programmer with the support facilities needed for low-level control of microprocessor components. The goal set up was to equally match the scope of control available at the level of assembly language programming.

The solution presented is based on an adaptable compiler system for the high-level systems programming language Modula-2. Given specifications concerning the implementation of operands, operators, and code sequences associated to special procedure calls the adaptable compiler system will be able to

- a) collect the information to be used by the front end of the Modula-2 compiler causing it to generate an intermediate representation with internal operators associated to the specified code generating sequences,
- b) generate a new code generator part able to recognize the internal operators and to execute the associated code generating sequence when the corresponding reduction is performed.

The compiler system consists of a fix front end part of the Modula-2 compiler and a processor for each of the three description modules. The implementor module contains the basic code emitting procedures and other specifications shared by all potential code generators for a specific processor family. The component modules contain the formal specifications of the implementation of operand types, operators, and special procedures. The configuration module defines the component objects to be included when the information for the front end and a code generator part for a new target configuration is generated. The BOBS parser generator is a common part of all the processors, the front end, and the generated code generator.

The specification technique is demonstrated by examples concerning the implementation of the functions available in a numeric coprocessor, process and interrupt transfer control, multi-processing mechanisms, protection and virtual management control, and some ideas of how the preprogrammed functions in firmware components can be utilized.

The main results can be summarized in the following points.

- 1) The adaptable compiler system will support the use of multiple precision data operands.
- 2) Any sequence of assembler instructions may be defined to be generated as a result of a high-level procedure call. Parameters may be passed for reference at the assembler code level. Code sequences may be optionally defined for inclusion in the run-time system.
- 3) A high-level interface between the user and the adapted compiler to be generated has been defined. The implementation decisions are specified in a notation close to that of a programming language.
- 4) The STANDARD/SYSTEM convention will separate the default implementation of data types, operators and standard Modula-2 procedures from the additionally defined objects thereby preserving the integrity of the language.

14.2 DISCUSSION

The translation technique used, i.e. syntax directed translation based on an attributed grammar, is far from new. It has been successfully used in numerous compilers especially in the analyzing front end part. The application of the technique in the code generating part in combination with an efficient LR-parser is more recent. In this thesis these experiences are applied to the problem of low-level control from a modern high-level programming language with a design that includes basic concepts for such control. The result achieved is that the extent of low-level control can be programmed to equally match that of assembly programming.

The main disadvantage of the method is that the programming of the code emitting code sequences in the operator and procedure definitions requires an understanding of the implementation model. The separation of the code generator part on an implementation level and a component definition level will to some extent compensate for this. The specifications in the component modules will in general require only basic knowledge of the register usage, expression evaluation and available non-terminal attributes. The more formal code rule

specifications in the component modules will also make it possible for the component module processor to ensure that the grammar rules are correct and compatible with the rest of the grammar. At this level errors will be possible to introduce only in the code emitted.

There is also a risk that the possibility to specify efficient system procedures may tempt the programmer to make too frequent use of this facility, i.e. the system procedures might be used as a kind of assembler routines opposing the objectives of Modula-2.

14.3 LIMITATIONS AND FUTURE WORK

The interest of this thesis has primarily been devoted to the iAPX86 processor family. Even if other processor families such as the NS16000, Z8000, MC68000 and TMS99000 have been studied alongside, the experience from the actual implementation of components from these families might have influenced the design decisions and maybe drawn the attention to problems now overlooked.

The general problems connected to retargetability such as efficient register allocation and optimization have not been discussed. The reason for leaving these subjects out is that they do not affect the discussion of the low-level control problem in principle, even if complicated register allocation mechanisms or optimization strategies makes it harder to program correct code emitting sequences in the component module specifications.

In the near future there are plans to make a full implementation of the adaptable compiler system and to include at least one of the other processor families apart from the iAPX86 family.

14.4 PRACTICAL APPLICATIONS

The application of the ideas and results of this work lies in the field of highly software/hardware integrated systems. Examples of such systems are interpreters, operating systems, process control and communication systems.

If the ongoing development of sophisticated processor components continues then the future applications will be even more interesting.

14.5 STATISTICS

An experimental implementation of a Pascal based compiler writing system supporting a Modula-2 cross compiler for the Intel iAPX86 processor family has been realized on the Vax-11/780 computer. The implementation efforts have been concentrated on the parts of interest for low-level control and some essential facilities for a complete system have not been implemented. Most of these limitations affect the front end part only. The implementor module supports close to a full implementation of Modula-2.

For the limited implementation of the front end of the Modula-2 compiler the following figures are valid.

Pascal source code	3000 lines
Executable code	40K bytes
Parser tables	30K bytes
Modula-2 grammar	250 productions

The implementor will be able to control the size and complexity of the code generator part. The size of the parser tables depends of course on the number of grammar rules included. The number of grammar rules depends not only on the number of components included but also very much on the ambition to produce efficient code. In the implementation model for the iAPX86 system three combinations of operands, i.e. $\langle \text{Expr} \rangle - \langle \text{Constant} \rangle$, $\langle \text{Expr} \rangle - \langle \text{Expr} \rangle$, and $\langle \text{Expr} \rangle - \langle \text{Var} \rangle$ have been considered in all operator specifications in order to take advantage of the memory addressing and immediate constant processing instructions in the 8086 processor. A simpler scheme had been possible to implement but also a more sophisticated.

The iAPX86 implementor module:

Support procedures	500 lines
Intermediate grammar	120 productions
Code rule source statements	400 lines

The iAPX86 processor component module:

Grammar rules	140 productions
Code rule source statements	600 lines

The resulting code generator occupies 2100 lines of Pascal code and 40K bytes of executable code. The parse table has a size of 20K bytes.

The BOBS parser generator occupies 3800 lines of Pascal source code and 40K bytes of executable code on the VAX-11/780 computer. For the skeletal parser the corresponding figures are 720 lines of Pascal source code and 30K bytes of executable code.

ACKNOWLEDGEMENTS

I greatly acknowledge the valuable comments from Professor Carl-Erik Fröberg and Göran Fries.

I am also grateful to Ralph Haglund for introducing me to the world of microprocessors, to Håkan Barregård for expanding my knowledge and for participating in the project, and to Thor-Björn Bladh for his support and for assembling and maintaining the microsystem.

REFERENCES

- [Aho 73] A.V. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling"; Vol. 1 and 2, Prentice-Hall, Englewood Cliffs, N.J., 1973.
- [Catt78] R.G.G. Cattell, "Formalization and automatic derivation of code generators"; Ph.D. dissertation, Computer Sciences Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.
- [Catt79] R.G.G. Cattell, J.M. Newcomer and B.W. Leverett, "Code generation in a machine-independent compiler"; In Proc. ACM SIGPLAN Symp. Compiler Construction (Denver, Colo., Aug.), ACM SIGPLAN Not. 14, 8, Aug. 1979, pp. 65-75.
- [Catt80] R.G.G. Cattell, "Automatic derivation of code generators from machine descriptions"; ACM Trans. Program. Lang. System 2, April 1980, pp. 173-190.
- [Erik82] S.H. Eriksen, B.B. Jensen, B.B. Kristensen and O.L. Madsen, "The BOBS System"; DAIMI PB-71 (Third edition), Aarhus Univ., Denmark, February 1982.
- [Gana80] M. Ganapathi, "Retargetable code generations and optimization using attribute grammars"; Ph.D. dissertation, Computer Science Dep., Univ. of Wisconsin-Madison, 1980.
- [Gana82] M. Ganapathi, N. Fischer and J.L. Hennessy, "Retargetable Code Generation"; In Computing Surveys, Vol. 14, No. 4, December 1982.
- [Glan78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation"; 5th Symposium on Principles of Programming Languages, 1978, pp. 231-240.
- [Grah80] S.L. Graham, "Table-driven code generation"; IEEE Computer, 13, 8, August 1980, pp. 25-34.
- [IBM 68] "IBM System/360 PL/I Language Specifications"; Form Y33-6003, IBM Corporation, 1968.

- [Ichb80] J.D. Ichbiah, et. al., 'Reference Manual for the Ada Programming Language', U.S. Department of Defense, July, 1980.
- [Jens75] K. Jensen and N. Wirth, 'Pascal User Manual and Report'; Springer Verlag, New York, 1975.
- [Jonn75] S.C. Johnson, 'YACC: Yet Another Compiler Compiler'; Techn. Report CSTR32 Bell Labs., Murray Hill, 1975.
- [Kern78] B.W. Kernighan and D.M. Ritchie, 'The C Programming Language'; Englewood Cliffs, N.J.: Prentice-Hall, 1978.
- [Knut68] D.E. Knuth, 'Semantics of context-free languages', Math. Syst. Theory, 2, June 1968, pp. 127-145.
- [Lune83] H. Lunell, 'Code Generator Writing Systems'; Ph.D. dissertation, Software Systems Research Center, Linkoping University, Sweden, 1983.
- [Moor74] C.H. Moore, 'FORTH: A New Way to Program a Computer'; Astronomy and Astrophysics Supplement, 15, 1974, pp. 497-511.
- [Wirt68] N. Wirth, 'PL360 - A Programming Language for the 360 Computers'; Comm. ACM, Vol. 15, No. 1, January 1968, pp. 37-64.
- [Wirt82] N. Wirth, 'Programming in Modula-2'; Springer Verlag, 1982. ISBN 0-387-11674-5.

The Syntax of the ACS

The syntactic definitions in the order of appearance in the thesis.

- 1 ComponentModule = COMPONENT MODULE ident ",",
 [export]
 {import}
 {ComponentSpec}
 END ident " ." .

- 2 ComponentSpec = StandardTypeSpec |
 SystemTypeSpec |
 StandardOpDecl |
 SystemOpDecl |
 StandardProcDecl |
 SystemProcDecl |
 SpacePrefixSpec .

- 3 StandardTypeSpec = STANDARD TYPE StandardTypeList.
- 4 StandardTypeList = StandardType {";" StandardType}.
- 5 StandardType = ident "=" BasicType [OF] number.

- 6 SystemTypeSpec = SYSTEM TYPE SystemTypeList.
- 7 SystemTypeList = SystemType {";" SystemType}.
- 8 SystemType = ident "=" BasicType [OF] number |
 ident "=" PointerType TO SystemType |
 ident "=" BitStrucType .

- 9 BitStrucType = PrefixType RECORD BitFieldList END.
- 10 BitFieldList = BitField {";" BitField}.
- 11 BitField = [BitIdent ":" SimpleType].
- 12 BitIdent = ident "[" "BIT:" ConstExpression.
 [".." ConstExpression] "]" .

- 13 StandardOpDecl = STANDARD OpDecl.
- 14 SystemOpDecl = SYSTEM OpDecl.
- 15 OpDecl = OPERATOR "(" Type Operator Type ")" ":" Type ";"
 [BEGIN
 OpRuleSequence]
 END .

- 16 OpRuleSequence = OpRule {; OpRule}.

```

17  OpCodeRule = RULE OpProduction [DO
                                StatementSequence]
                                END.
18  OpProduction = NonTerminal "::~="
                                [NonTerminal] Operator NonTerminal.
19  NonTerminal = "<" ident ["." number] ">".
20  StandardProcDecl = STANDARD PROCEDURE standardprocident
                                [(" TypeParamList ")
                                [":" Type]]
                                [BEGIN
                                ProcRuleSequence]
                                END.
21  TypeParamList = TypeParam {"," TypeParam}.
22  TypeParam = [VAR | CONST] [ident ":" ] Type.
23  standardprocident = ABS | CAP | CHR | FLOAT ` | HIGH | ODD |
                                ORD | TRUNC | VAL | DEC | EXCL | HALT |
                                INC | INCL.
24  ProcRuleSequence = ProcCodeRule {";" ProcCodeRule}
25  ProcCodeRule = RULE ProcProduction [DO
                                StatementSequence]
                                END.
26  ProcProduction = NonTerminal "::~="
                                standardprocident [(" NonTerminalList ")].
27  NonTerminalList = NonTerminal {"," NonTerminal}.
28  SystemProcDecl = SYSTEM PROCEDURE ident
                                [(" TypeParamList ") [":" Type]]
                                [BEGIN
                                ProcRuleSequence]
                                END.
29  SpacePrefixSpec = SYSTEM SPACE ident "=" ident OF SubrangeType.
30  ImplementorModule = IMPLEMENTOR MODULE ident ";"
                                [export]
                                {import}
                                {declaration}
                                {TerminalSpec}
                                {AttributeSpec}
                                {ImplCodeRuleSequence}
                                {RunTimeProcedure}
                                [BEGIN
                                StatementSequence]
                                END ident "." .

```



```

31  AttributeDecl = CommonAttributeDecl | PrivateAttributeDecl.
32  CommonAttributeDecl = COMMON ATTRIBUTE ident "=" SimpleType.
33  PrivateAttributeDecl = PRIVATE ATTRIBUTE NonTerminal "="
                                RecordType.

34  TerminalSpec = TERMINAL TerminalList.
35  TerminalList = Terminal {"," Terminal}.

36  ImplCodeRuleSequence = ImplCodeRule {";" ImplCodeRule}.
37  ImplCodeRule = RULE ImplProduction [DO
                                StatementSequence]
                                END.

38  Production = NonTerminal "::~=" SymbolList.
39  SymbolList = Symbol {Symbol}.
40  Symbol = NonTerminal | Terminal | EMPTY.

41  RunTimeProcedure = RUNTIME PROCEDURE ident ";"
                                BEGIN
                                `(*` AssemblerStatements `*)`
                                END ";" .

42  ConfigurationModule = CONFIGURATION MODULE ident ";"
43                        {import}
44                        END ident "," .

```

The following non-terminals appearing on the right hand side in the production rules are defined according to the standard Modula-2 syntax. The number refers to the ordering in the syntax list in [Wirt82].

ident	1
number	2
ConstExpression	13
Type	24
SubrangeType	29
PointerType	40
StatementSequence	59
export	86
import	87

